

Calhoun: The NPS Institutional Archive

DSpace Repository

Reports and Technical Reports

All Technical Reports Collection

1981-03

The Naval Postgraduate School secure archival storage system, Part II: Segment and process management implementation

Schell, Roger R.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/29013

Downloaded from NPS Archive: Calhoun



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School 411 Dyer Road / 1 University Circle Monterey, California USA 93943 NPS52-81-001



NAVAL POSTGRADUATE SCHOOL

Monterey, California



The Naval Postgraduate School SECURE ARCHIVAL STORAGE SYSTEM

Part II

- Segment and Process Management Implementation -

Lyle A. Cox, Roger R. Schell, and Sonja L. Perdue

March 1981

Approved for public release; distribution unlimited

red for:

FEDDOCS D 208.14/2:NPS-52-81-001

of Naval Research gton, Virginia 22217



NAVAL POSTGRADUATE SCHOOL Monterey, California

Rear Admiral J. J. Ekelund Superintendent

D. A. Schrady Acting Provost

This research was partially supported by grants from the Office of Naval Research, Project No. 427-001, monitored by Mr. Joel Trimble, and the Naval Postgraduate School Research Foundation.

Reproduction of all or part of this report is authorized.

This report was prepared by:

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
	. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
NPS52-81-001 4. TITLE (and Substite)		5. TYPE OF REPORT & PERIOD COVERED
The Naval Postgraduate School		Technical Report
SECURE ARCHIVAL STORAGE SYSTEM	goment	
Part II - Segment and Process Mana Implementation	gement	6. PERFORMING ORG, REPORT NUMBER
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(#)
Lyle A. Cox, Roger R. Schell, and Sonja L. Perdue		
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Naval Postgraduate School Monterey, CA 93940		61152N; RR000-01-10 N 000 1481WR10034
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE
Naval Postgraduate School Monterey, CA 93940		March 1981
		451
14. MONITORING AGENCY NAME & ADDRESS(II different I	rom Controlling Office)	15. SECURITY CLASS. (of thie report)
Chief of Naval Research Arlington, Virginia 22217		Unclassified
All Higgsii, Virginia 22217		15. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abetract entered in	Block 20, if different from	n Report)
18. SUPFLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse elde if necessary and	identify by block number)	
Security Kernel, Microcomputers, A Operating Systems, Computer Securi	rchival Storage ty	, Computer Networks,
20. ABSTRACT (Continue on reverse side it necessary and in	dentify by block number)	
The security kernel technolog for highly reliable protection of operating system implementations f (1) adequate computational resourc clean, straightforward structure wathis paper presents the experience ation using the Advanced Micro Dev	computerized in ace two signifi es for applicat hose correctnes on an ongoing	formation. However, the cant challenges: providing ions tasks, and (2) a s can be easily reviewed. security kernel implement-

DD 1 JAN 73 1473

EDITION OF | NOV 65 IS OBSOLETE S/N 0102-014-6601 |

Unclassified

LLURITY CLASSIFICATION OF THIS PAGE(When Date Entered)

20.

the Z8002 microprocessor. The performance issues of process switching domain changing, and multiprocessor bus contention are explicitlyly addressed. The strictly hierarchical (i.e., loop-free) structure provate series of increasingly capable, separately usable operating system subsets. Security enforcement is structured in two layers: the basic kernel rigorously enforces a non-discretionary (viz., lattice model) policy, while an upper layer provides the access refinements for a discretionary policy.

The Naval Postgraduate School SECURE ARCHIVAL STORAGE SYSTEM

Part II

-Segment and Process Management Implementation-

by

Roger R. Schell, Lyle A. Cox, and Sonja L. Perdue March 1981

Report number: NPS52-81-001

THE STRUCTURE OF A SECURITY KERNEL FOR A Z8000 MULTIPROCESSOR

LYLE A. COX, Jr., and ROGER R. SCHELL, Col., USAF

Department of Computer Science

Naval Postgraduate School

Monterey, California

ABSTRACT

The security kernel technology has provided the technical foundation for highly reliable protection of computerized information. However, the operating system implementations face two significant challenges: providing (1) adequate computational resources for applications tasks, and (2) a clean, straightforward structure whose correctness can be easily reviewed. This paper presents the experience of an ongoing security kernel implementation using the Advanced Micro Devices 4116 single-board computer based on the Z8002 microprocessor. The performance issues of process switching, domain changing, and multiprocessor bus contention are explicitly addressed. The strictly hierarchical (i.e.,

loop-free) structure provides a series of increasingly capable, separately usable operating system subsets. Security enforcement is structured in two layers: the basic kernel rigorously enforces a non-discretionary (viz., lattice model) policy, while an upper layer provides the access refinements for a discretionary policy.

BACKGROUND

For the last two and a half years the Naval Postgraduate School has been conducting a research and development project involving security kernel based operating systems designed for multiple processor implementations. As this work continues we feel that it is important to report on our progress and experiences, especially in the area of microprocessor implementations.

This effort has come to be known as the "SASS" or Secure Archival Storage System project [1]. In fact, this is a misnomer, as SASS is but a single instance of a more general family of secure operating systems designed early in the project [2]. While SASS has been the object of the majority of the research reported it is not the only implementation. Another operating system of this family has also been written to support a signal processing system that uses multiple Intel 8086 processors [3].

SASS has been our principal testbed for exploring the implementation and performance issues related to these types of operating systems. SASS itself was designed to be a comprehensive multiuser, multilevel secure file storage system. designed, it will consist of a small number of Z8000-based [4] single board computers sharing a single Multibus with storage devices and input/output devices. will interface via bidirectional lines to a number of "host" systems, as illustrated in Figure 1. SASS will provide each host with a hierarchical file system. This system can be used to store and retrieve files, and share files with other hosts. This design will allow SASS to serve as a central hub of a data secure network of computers with diverse security authorization for sensitive information. vides archival, shared storage while insuring that each interfaced host processor can access only that information appropriate to its security authorizations.

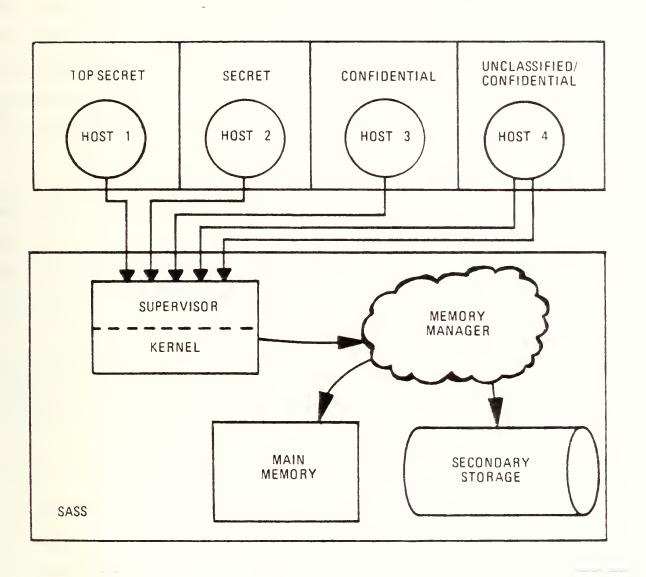


Figure 1: SASS System Interfaces

STRUCTURE

For this family of operating systems the security kernel technology has been used not only to effect security but also to provide the underlying organizational framework for the operating system. The SASS, one member of this family, is in the final stages of implementation. This development experience has highlighted the importance of several features that are key to this family:

- The pervasive, yet systematizing impact of the security kernel methodology [5].
- The design simplicity that accompanies a loop-free modularization that is highly compatible with the resource sharing and multiprogramming functions.
- The significance of a high degree of configuration independence, particularly for the ability to use the latest microprocessors for testbed implementation.

Independent of security, this particular kernel structure is attractive as a canonical operating system interface. It appears adequate for a wide range of functionality and capacity, and it evidences a high degree of independence from hardware idiosyncrasies. These operating system features will be discussed further below.

Security Kernel Approach

Members of this operating system family are organized with three distinct extended machine layers: (1) the security kernel, (2) the supervisor, and (3) the applications. This is illustrated in Figure 2. The concept of a hierarchy of extended machines is, to be sure, not new; however, the security kernel significantly constrains the organization. In particular, for reason of security all the management of physical resources must be within the kernel itself. Furthermore, confidence is increased by keeping the kernel as small and simple as possible. This means that much of what is commonly thought of as the operating system is provided outside the kernel in the supervisor layer. For this particular family member there is no major applications layer (viz., within SASS itself), since the applications are contained in the individual hosts.

The basic family of operating systems requires the kernels to provide extended virtual machines that specifically support both asynchronous processes and segmented address spaces. Within SASS, the kernel virtualizes processors, all levels of storage, and input/output. The kernel creates virtualized objects -- processes, segments, and devices. It is this "pure" virtual interface that is attractive as the

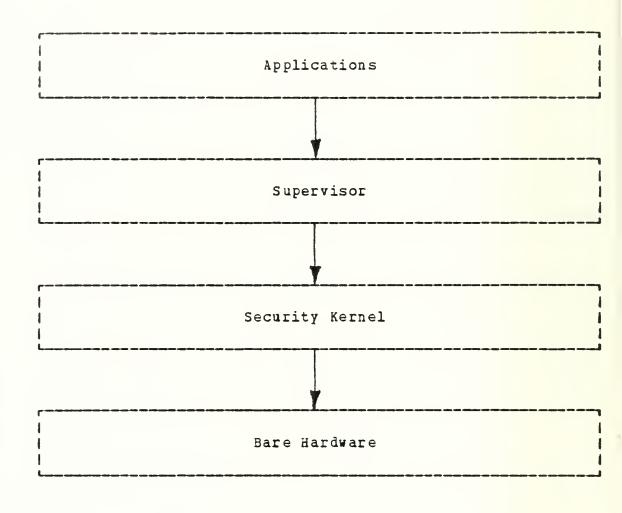


Figure 2: Extended Machine Layers

basis for canonical operating system features. The SASS supervisor is in turn built upon the kernel, using these virtualized objects to construct the file system.

Both the kernel and the supervisor have certain responsibilities for system security. The kernel manages all physical resources, and the kernel is distributed (i.e., included) in the address space of every process. At this level, isolation of the kernel -- protection from users and the supervisor -- must be provided by hardware enforced domains. The design of the system is strictly hierarchical (viz., the kernel is more privileged than the supervisor) so protection rings, as defined for Multics [6], are a satisfactory domain implementation.

The kernel has the responsibility for the enforcement of access limitations: that is, the kernel provides the mechanism for supporting non-discretionary security policy. The SASS kernel can support any such policy which can be expressed by a lattice of access classes [7]. Every object -- process, segment, or device -- has a non-forgable label that denotes its access class. This non-discretionary security has been parameterized in SASS such that exactly one module has knowledge of the interpretation of this label in terms of a specific policy. Thus, only this single module need be tailored to support a particular policy.

SASS provides discretionary security (shared access within the bounds of non-discretionary policy based on individual user identification) via the supervisor and the file structure. This discretionary security is completely outside of the kernel (in contrast with the KSOS [8] approach).

The supervisor handles the "Secure Reader-Writer Problem" with a non-exclusionary approach (one writer, retry on read) to provide synchronization between processes of different access classes. This control of interprocess communication is implemented via kernel primitives using Reed's event-counts and sequencers [9].

The SASS supervisor capabilities are achieved by associating two processes with each host link. These processes access that portion of the SASS file structure associated with that host. One of these processes provides I/O transmission and link management, while the other, a file manager, is responsible for the file system structure of its associated host. Communication between these processes (as is communication between all processes) is achieved using shared segments -- a mailbox. Synchronization is provided by the kernel (with eventcounts and sequencers).

The complementary kernel/supervisor approach to security has several advantages for SASS: the size and the complexi-

ty of the kernel can be minimized, and, given reliable host authentication, any host weaknesses will not impact the reliable enforcement of the non-discretionary security policy.

The security kernel approach constrains not only the interface but also the detailed design and implementation of internal state variables. The problem is to prevent indirect information paths between processes with different access classes. We address this problem using essentially the approach detailed by Millen [10], although without the rigor of a proof. Internal state variables, e.g., shared resource tables, are assigned an access class, and it is confirmed that its values will not be reflected to processes of an inconsistent access class. The most apparent result is that the "success code" (returned in response to the invocation of kernel primitives) primarily reflects the state of the per-process virtual resources, not the shared physical resources.

Loop Free Organization

Another aspect of the design that has helped to keep the security kernel simple and understandable is the loop-free structure of SASS. The loop-free design supports the software engineering concept of "information hiding" [11], as there are really no global data structures within SASS. The kernel is internally organized into four distinct layers, as illustrated in Figure 3; these layers, that will be described below, are termed (1) segment and event managers, (2) traffic controller, (3) memory manager, and (4) inner traffic controller.

In practice we have been quite doctrinairs in enforcement of the loop-free structure for this organization. While many operating systems claim to be modular or well-structured, we empirically validate this claim. We "peel-off" the upper layers one at a time by literally removing the code and data, and then demonstrate that the remainder can be loaded and run as a functionally intact, but obviously limited, operating system subset. The function of each layer will now be described, proceeding from the bottom upward.

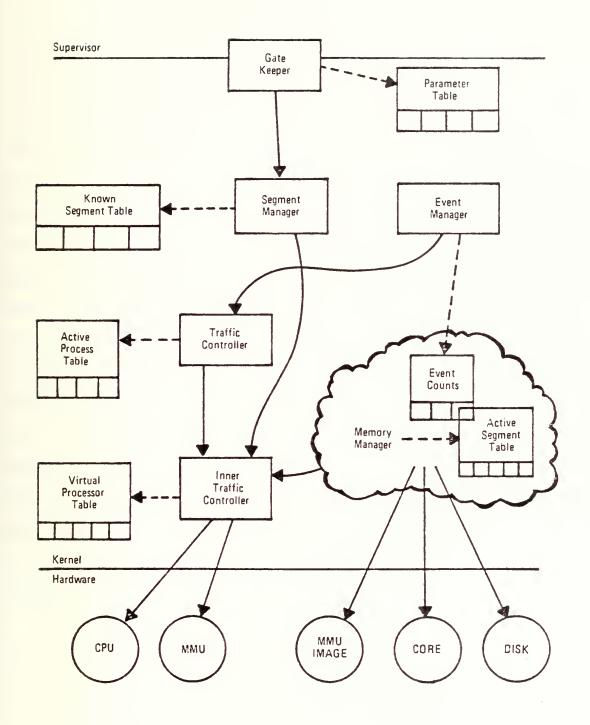


Figure 3: Internal Kernel Organization

Inner Traffic Controller. Processor multiplexing has two layers, similar to those proposed for Multics [12]. Each physical processor has a fixed number of "virtual processors" that are multiplexed onto it. Two of these virtual processors are dedicated to system services: an idle virtual processor and a memory manager process to manage the asynchronous access to secondary storage devices. The remaining virtual processors (currently two per physical processor) are available to the (upper level) traffic controller. The inner traffic controller provides signal and wait synchronization primitives that include a message that is passed between virtual processors. In terms of traditional jargon, the inner traffic controller provides multiprogramming by scheduling virtual processors to run on the CPU they are (permanently) associated with. Note that this structure implies that the security kernel is interruptible, viz., is not a critical section; however, the inner traffic controller itself is not interruptible. In addition, this layer provides all the multiprocessing interactions between individual physical processors, using a hardware "preempt" interrupt.

Memory Manager. This layer manages the multiplexing of the physical storage resources, viz., "disk" and "core". This layer also manages the segment descriptors in the memory management unit (MMU) image for each process. Most of the functions of this layer are executed by the per-CPU memory

manager processes, with synchronization provided by inner traffic controller signal and wait primitives. The single board computers have per-processor, local memory; there is also additional global memory that is addressable by all processes. The memory manager insures that (only) shared segments are in global memory.

This policy can require some transfers between local and global memory; however, the low transaction rate of the archival storage system is not demanding, and this structure minimizes bus transfer requirements under expected operating conditions.

Traffic Controller. The variable number of processes (two per host) are multiplexed onto virtual processors defined by the inner traffic controller. Each process has an affinity to the physical processor whose local memory contains a portion of its address space at the time of the process scheduling decision. As indicated earlier, the traffic controller layer uses Reed's advance and await mechanism [9] to provide interprocess communication.

<u>Segment and Event Managers</u>. All entries into the kernel pass through the segment/event manager layer. The explicit non-discretionary security checks are made at this level by comparing the access class labels of subjects and objects. This layer uses a per-process known segment table to convert

process local names (viz., segment number) for objects into system-wide names. Each segment has associated with it two eventcounts and a sequencer; thus, segment numbers also serve as their names. The segment manager provides for the creation and deletion of segments and their entry into and removal from a process address space.

Gate Keeper. A process invokes a security kernel function using the traditional trap mechanism. The Z8000 "system call" instruction causes a trap, and the gate keeper is merely the trap handler. All parameters and return values are "passed by value" in CPU registers; this simplifies security validation. The gate keeper merely calls the particular procedure that corresponds to the requested function.

Microprocessor Testbed

One important aspect of this research has been the actual implementation and testing of the concepts developed. Traditionally the implementation of multiple processor structures has been an expensive undertaking. Recently the development of sophisticated microprocessors that feature multiple operating modes, advanced addressing, support of multiple processor configurations, and a standard bus configuration with peripheral support have all made the implementation of advanced operating systems on microprocessor devices possible, and economically feasible.

The processors of SASS all share the same bus; each processor is a commercial single board computer with on-board random access memory. These processors also share a global memory, and certain peripheral devices. This configuration is illustrated in Figure 4.

In general, security kernel based operating systems find three processor-supported execution domains (operating states) highly desirable: for the kernel, supervisor, and applications. This is true of the operating system family discussed here. Currently there are no single chip processors that support three states. This is not a significant problem for SASS, since it is the hosts rather than the SASS system processors that execute user application programs. Under these circumstances a two mode (kernel and supervisor) machine is sufficient. Such architectures are currently available as microprocessors, in particular the 28000.

Accordingly, we are implementing a multiple microprocessor system to test the SASS concept. The current hardware in use is the AMD 4116 single board computer [13] in a standard Multibus backplane. This configuration has a significant limitation: it does not include the hardware Memory Manager Unit, as described in [2].

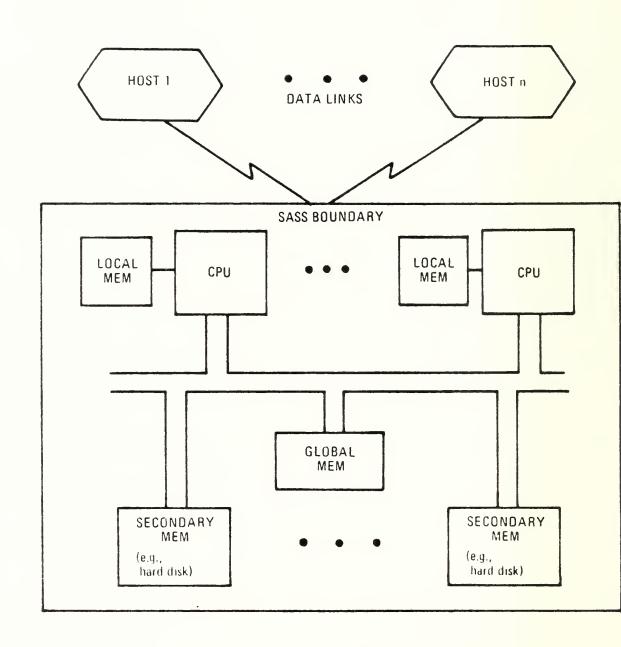


Figure 4: Multiprocessor Configuration

Currently we simulate in software the memory management unit, so the kernel is not protected from the supervisor as the original design specified. Hardware protection in the form of addressing limitations is available, and has been used in some of the experiments to assure the integrity of the kernel. In this configuration, the hardware protects one half of the local memory from any access when the CPU is operating in the normal mode. Any attempt to access memory which is thus protected generates an interrupt and the fault detection software traps the access. This is adequate for current tests, but a complete memory management system is clearly more desirable. Our experiences on this testbed in terms of performance and software development are discussed further below.

THE SASS EXPERIENCE

The lessons learned to this point fall into two broad categories: programming (software engineering) experiences, and performance experiences. We will discuss both of these issues below.

Programming Experiences

The nature of this research effort has been highly structured, emphasizing modularity at every opportunity. The software design is strictly "top-down". This has been a

matter of good design practice, and of necessity. Since the majority of the work has been performed by a succession of Master's degree students [14,15,16,17,18,19] during their brief six to nine months of research each, the clear definition of software modules has been key to the success of the effort. We have found that the high degree of modularity has allowed the students to work on the project with a minimum of "start-up" time, and a maximum of productive effort and learning.

The actual implementation is proceeding in an essentially bottom up manner, with test harnesses and stubs being written as necessary for testing. The SASS modules were specified in a pseudo-language resembling current higher level languages. The SASS modules as implemented were coded in PLZ-ASM [20], the Z8000 structured assembly language. We found that the pseudo-code specifications of modules were adequate, and that the translation from this code to the structured assembly language was straightforward.

The structured assembly language of the Zilog Z8000 supported many of the constructs usually thought to be unique to higher level languages, including typed record structures, DO-loops, IF-THEN-ELSE, and CASE. In fact, our programmers think of this assembly language as a higher level language. Approximately 40 percent of the statements writ-

ten in SASS are equivalent to statements in modern programming languages.

Despite the qualities of the structured assembler, it was selected by default. When the decision was made, the prototype hardware boards were just becoming available. There was virtually no software support available. In particular, no higher level language was available. The software environment was (by modern standards) very primitive, with no tools for operating system development available. Nevertheless, the progression from microprocessor development system to commercial single board computer system has been surprisingly smooth (an opinion that some students might dispute). The software development environment has grown slowly. Yet, this has not proved to be a handicap.

Performance Issues

In the programming for the SASS, we have generally treated performance as a secondary issue, in deference to more basic concerns such as security and modularity. However, we have addressed performance on a design level where performance is strongly related to architectural choices.

Obviously, one basic design choice is the use of multiprocessing as a way to increase processing capacity. However, bus contention is a major performance concern in the
multiprocessor configurations, since all processors share a
single Multibus. If, for example, all code and data were
located in global memory, then even two or three processors
would saturate the bus. However, in reality only shared,
writable segments need be in global memory. Our use of a
purely virtual, segmented memory permits the kernel to determine exactly which are the shared, writable segments. As
noted before, the memory manager layer totally controls the
allocation to global memory, and thus markedly controls bus
contention.

In the current SASS implementation we use the "Normal" and "System" modes of the Z8000 hardware, with the system mode dedicated to the security kernel. The domain changes automatically generate a switch of the stack within the

hardware. This is particularly important to the efficiency with which we can switch domains while maintaining the integrity of the kernel.

In SASS a process switch is achieved by switching the stack. SASS saves the process history in the stack, so a switch requires only the stack exchange. Preempt hardware interrupts can initiate scheduler changes, and associated virtual interrupts to the virtual processors. This sequence is relatively efficient given the Zilog architecture. The process switching performance question is more interesting in the context of processor multiplexing.

The multiprogramming time is the interval from the time the inner traffic controller signal primitive is invoked in one virtual processor until there is a return from a (pending) wait invocation in a different virtual processor. This includes both process switching and message passing operations.

For interprocess communication, the read and ticket calls (from the normal mode) include a system call though the gate keeper to the kernel, the non-discretionary security checks, and access to the eventcount or sequencer value; however, no process switch is involved. The synchronization time includes the interval from the invocation of the system call

(in normal mode) for advance in one process until the return from a (blocking) await invocation in a different process.

This includes the security checks and scheduling of both a virtual and a physical processor.

A set of measurements on the current implementation are summarized in Table 1. There has been no effort to "tune" the system to improve performance. We find these results within our range of expectations for a single chip microprocessor.

<u>Function</u>	Time (milliseconds)
Multiprogramming signal/wait pair	0.5
Synchronization advance/await pair	2.3
Read (Eventcount)	0.6
Ticket (Sequencer)	0.6

Table 1. Performance Measurements

SUMMARY

A modern operating system featuring kernel based security, segmented memory and multiple processors has been designed and is being implemented using modern microprocessors. To date our focus on methodical design has paid off: the implementation of a carefully designed, simple structure using elementary software development tools has proceeded well.

The initial testbed implementation is running and preliminary data is now available regarding the operating performance of such systems implemented on microprocessors of advanced architectures. Data gathered suggests that the security kernel is indeed an attractive structure for a modern operating system. There is a wide range of applications where sophisticated operating systems can be implemented upon microprocessors, and attractive performance can be achieved, particularly through the use of multiple processors.

<u>ACKNOWLEDGMENTS</u>

The authors would like to acknowledge the many long hours of work and dedicated effort contributed by E. E. Moore, A. V. Gary, S. L. Reitz, J. T. Wells, and A. R. Strickler, the students of the SASS project. Without their dedication, ideas and effort, this project would never have been able to progress. Specifically, we would like to acknowledge the contributions of Ms. C. Yamanaka and Ms. N. Seydel, whose typing and assistance were invaluable. This research was partially supported by grants from the Office of Naval Research, Project No. 427-001, monitored by Mr. Joel Trimble, and the Naval Postgraduate School Research Foundation.

REFERENCES

- [1] Schell, R. R. and Cox, L. A. "A Secure Archival Storage System," PROCEEDINGS OF COMPCON FALL, September 1980.
- [2] O'Connell, J. S. and Richardson, L. D., <u>Distributed</u>,

 <u>Secure Design for a Multi-microprocessor Operating Sys-</u>
 <u>tem</u>, Master of Science Thesis, Naval Postgraduate
 School, June 1979.
- [3] Schell R. R., Kodres, U. R., Amir, H., Wasson, J., and Tao, T. P., "Processing of Infrared Images by Multiple Microcomputer System, Proceedings SPIE Symposium, "Real-Time Signal Processing III," Vol. 241, (1980), pp. 267-278.
- [4] Peuto, B. L., "Architecture of a New Microprocessor,"

 Computer, Vol. 12, No. 2, February 1979, p. 10.
- [5] Schell, R. R., "Security Kernels: A Methodical Design of System Security," USE Technical Papers (Spring Conference, 1979), March, 1979, pp. 245-250.
- [6] Schroeder, M. D. and Saltzer, J. H., "A Hardware Architecture for Implementing Protection Rings," <u>Communications of the ACM</u>, Vol. 15, No. 3, March 1972, pp. 157-170.
- [7] Denning, D. F., "A Lattice Model of Secure Information Flow," Communications of The ACM, Vol. 19, May 1976, pp. 236-242.

- [8] McCauley E. J. and Drongowski, P. J., "KSOS The Design of a Secure Operating System," Proceedings of the National Computer Conference, 1979, pp. 345-371.
- [9] Reed, P. D. and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," <u>Communications of the ACM</u>,
 Vol. 22, No. 2, February, 1979, pp. 115-124.
- [10] Millen, J. K., "Security Kernel Validation in Practice," Communications of the ACM, Vol. 19, No. 5, May 1976, pp. 243-250.
- [11] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," <u>Communications of the ACM</u>,

 Vol. 15, No. 12, December 1972, pp. 1053-1058.
- [12] Schroeder, M. D. et. al., "The Multics Kernel Design Project," Proc. Sixth ACM Symposium on Operating Systems Principles, November 1977, pp. 43-56.
- [13] Advanced Micro Devices, Am 96/4116, Am Z8000 16-Bit MonoBoard Computer User's Manual, 1980.
- [14] Parks, E. J., <u>The Design of a Secure File Storage System</u>, Master of Science Thesis, Naval Postgraduate School, December 1979.
- [15] Coleman, A. R., <u>Security Kernel Design for a Micropro-</u> <u>cessor-Based</u>, <u>Multilevel Archival Storage System</u>, Mas-

- ter of Science Thesis, Naval Postgraduate School, December 1979.
- [16] Moore, E. E. and Gary, A. V., <u>The Design and Implementation of the Memory Manager for a Secure Archival Storage Systems</u>, Master of Science Thesis, Naval Postgraduate School, June 1980.
- [17] Reitz, S. L., An Implementation of Multiprogramming and Process Management for a Security Kernel Operating System, Master of Science Thesis, Naval Postgraduate School, June 1980.
- [18] Wells, J. T., <u>Implementation of Segment Management for</u>

 <u>a Secure Archival Storage System</u>, Master of Science

 Thesis, Naval Postgraduate School, September 1980.
- [19] Strickler, A. R., <u>Implementation of Process Management</u>

 <u>for a Secure Archival Storage System</u>, Master of Science

 Thesis, Naval Postgraduate School, March 1981.
- [20] ZILOG, Inc. Z8000 PLZ/ASM Assembly Language Programming Manual, April 1979.

FOREWORD

This technical report contains edited segments of four masters theses:

The Design and Implementation of the Memory Manager for a Secure Archival Storage System by E. E. Moore and A. V. Gary

An Implementation of Multiprogramming and Process
Management for a Security Kernel Operating System
by S. L. Reitz

Implementation of Segment Management for a Secure Archival Storage System by J. T. Wells

Implementation of Process Management for a Secure archival Storage System by A. R. Strickler

which describe the development and implementation of the Naval Postgraduate School Secure Archival Storage System (SASS). These theses are based upon the design outlined in the Naval Postgraduate School SECURE ARCHIVAL STORAGE SYSTEM Part I - Design - by R. R. Schell and L. A. Cox [17]. This design is updated and presented in detail.

Some sections of each thesis have been excluded in order to eliminate repetition and bulk. Similarly, the program listings in this report represent the current state of the project and do not pertain to any one thesis. An attempt has been made to footnote some discrepancies between the

However, there may be some details described herein which do not correspond to the current SASS system. Consequently, the reader is advised to consult the individual thesis if more detail on a particular phase of the development is required. A program description document, providing greater clarification of SASS organization and listings, is also available.

CONTENTS

THI	2	STR	JCI	'U'	RE	0	F	A	S	E	CÜ	RI	T	Z E	E	RNE	L	FO	R	A	Z8	00)						
					MU	LT	IF	R	OC	E	SS	OR	•				•	•	•	•	•	•	•	•	•	•	•	•	ii
POI	RE	WORI)	•	•	•	•		•	•	•	•	•				•	•		•	•			•				X	xx
										-	_																		
									PA	K	T	A	-		1	LNT	RC	DU	CT	10	N								
<u>Ch</u> a	<u>1</u> p	<u>ter</u>																										<u>p</u> <u>a</u>	<u>qe</u>
I.	В	ACK	GRO	Ū.	N D	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	2
II.	•	BASI	C	C	ИО	CE:	PΤ	s,	/D	E	ΡI	NI	T I	:01	IS	•	•	•	•	•	•	•	•	•	•	•	•	•	6
			PR	101	CE	SS					•															•			6
			IN	P) R	MA	TI	0	N	S	EÇ	UR	IJ	Y									•			•			8
			SE	G.	ME	NT.	ΑT	I	ON	ſ	•	•					•	•	•		•	•		•	•	•		•	13
						CT					MA	IN	S				•	•	•	•	•	•	•	•	•	•	•	•	15
			AB	S	ΓR	AC'	ΤI	0	N	•	•	•	•	•			•	•	•	•	•	•	•	•	•	•	•	•	16
		PAR	RT	В				S	EC	. 0	RE	A	RC	:H1	AS	1 L	SI	OR	AG	E	SY	ST	em	DE	2S]	[G N	i		
<u>Ch</u> a	<u>p</u>	<u>ter</u>																										<u>p</u> <u>a</u>	<u>qe</u>
		ter BAS	SIC	:	S A	SS	0	V.	ER	8 A	IE	W	•				•	•	•	•	•	•	•	•	•	•	•	<u>pa</u>	18
III							•) V .	er •	• V	IE.	\ W .	•			• •	•	•	•	•	•	•	•	•		•		<u>pa</u>	
III		BAS	er v	Ί	SG	R	•		•	•	•	•	•			•	•	•	•	•	•	•	•	•	•	•	•	<u>pa</u>	18 23
III		BAS	ZR V FI	I:	SO E	R MA	N A	.G	• ER	•	• PR	oc	ES			• •	•	•	•	•	•	•	•	•	•	•	•	<u>pa</u>	18 23 24
III		BAS	ZR V FI	I:	SO E	R	N A	.G	• ER	•	• PR	oc	ES			•	•	•	•	•	•	•	•	•	•	•	•		18 23
III	[•	BAS	ZR V FI IN	I.	SO E UT	R MA: /O	N A	.G	• ER	•	• PR	oc	ES				•	•	•	•	•	•	•	•	•	•	•		18 23 24
III	G G	BAS	ER V FI IN KE	I: P!	SO E UT PE	R MA /O R	· NA UT	G.	• ER UT		PR PR	oc	ES				•	•	•	•	•	•	•	•	•	•	•		18 23 24 25
III	G G	BAS SUPI	ERV FI IN KE	IL:	SC E UT PE	R MA /O R ED	NA UT	(G)	ER UT	E	PR PR	oc	ES				•	•	•	•				•	•			<u>pa</u>	18 23 24 25 26 28
III	G G	BAS SUPI	ERV FI IN KE	IL:	SO E UT PE UT	R MA /O R ED	NA UT K	G:P	ER UT RN	E	PR PR L	oc oc	ES	ss			•	•	•	•	•		•	•	•	•	•	<u>pa</u>	18 23 24 25 26 28 29
III	G G	BAS SUPI	FI IN KE	I: IP IE: IB	SO E UT PE UT ME	R MA /O R ED NT M	NA UT K	G:P	ERUT	EGR	PR PR	oc	ES	ss	UEUE	·	· · · · · · · · · · · · · · · · · · ·					•			•		• • • • • • • • • • • • • • • • • • • •		18 23 24 25 26 28
III	G G	BAS SUPI	FIIN KE	I E E E E E E E E E E E E E E E E E E E	SC E UT PE UT ME	R MA: /O R ED NT M: ISO	NA UT K	G:P	ERUT	EGRO	PR PR L ER	OC OC	ES	ss	UF	RIT	Y					•			•			<u>p</u> a	18 23 24 25 26 28 29 32 33
III	G G	BAS SUPI	FI IN KE CRI SE EV NO	I E G G A I A I A I A I	SC E UT PE UT ME NT-D	R MA /O R ED NT M	NA UT K	G:E	ERUT RN RA GETI	EGROR	PR PR	OC OC	ES	SEC			•		· · · · · · · · · · · · · · · · · · ·	LE		•			•		•	<u>p</u> a	18 23 24 25 26 28 29 32 33 33 33
III	G G	BAS SUPI	ERV FI IN KE CRI SE EV NO TR	I E G G I E I N I N I N I N I N I N I N I N I N	SC E UT PE UT ME NT PF FF	R MA: /O R ED NT M: IS	NA UT K M AN CR CR	G E A A A A A A A A A A A A A A A A A A	ERUT RN NA GE TI	EGRORC	PR PR L ER NA OL	OC OC	ES ES	SS EC	LE	e R	•	•		•	•	•		•			• • • • • • • • • • • • • • • • • • • •	<u>p</u> a	18 23 24 25 26 28 29 32 33

VI	I.	N	101	1 – D	I	ST	RI	В	UT	ED)	ΚE	RN	E	L	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	43
				ME	M	O R	Y	M	A N	ΑG	E	R	PR	0	CE	SS	;	•	•	•	•	•	•	•	•	•	•	•	•	•	43
VI	II	•	Sì	ST	E	M	HA	R	D W	AR	E	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	48
IX	•	St	JMN	1AR	Y	•	•		•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	52
	PA	RI	? (: MA																	TA								10E	RY	
<u>Ch</u>	<u>a p</u>	<u>t</u> e	Ē																											<u>p</u> :	<u>age</u>
х.	I	r n	RC	DU	C'	ΤI	ON	Ī	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•		•	•	•	•	•	54
XI	•	ME	emo	RY	i	MA	NA	G	ER	P	R	oc	ES	S	D	ΕI	'A	IL	ED	D	ES	IG	N	•	•	•	•	•	•	•	57
																					•								•	•	57
																					NS										
				DA	T	A	BA	S	ES		•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	63
																															63
						Lo	ca	.1	A	ct	i	ve	S	е	gm	en	t	T	ab	le		•	•	•	•	•	•	•	•	•	68
						A1	ia	S	T	ab	1	9	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	69
																															71
																					ti									•	
				BA	S	IC	P	U,	NC	TI	01	NS	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	75
					(CI	ea	t	9	an		Al	ia	s	T	a b	1	е	En	tr	Y	•	•	•	•	•	•	•	•	•	78
																															80
					i	A C	ti	V	at	е	a	S	eg	m	en	t	•	•	•	•	•	•	•	•	•	•	•	•	•	•	83
																															87
						SW	ap) ;	a	Se	gı	ne	nt	,	In		•	•	•	•	•	•	•	•	•	•	•	•	•	•	91
																					•										
																															98
						OE	٧e	1	a	Se	gi	пe	nt		to	G	1	do	al	. M	еm	or	Y	•	•	•	•	•	•	•	99
																															101
					1	ЦÞ	da	t	е	th	е	M	MU		Im	a g	je	•	•	•	•	•	•	•	•	•	•	•	•		102
				SU	M	MA	RY	•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		103
XI	ı.	S	TA	TU	S	0	F	R	ES	EA	R	СН	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		105
				CO	N (CI.	US	I	ON	S																					105
				FO																	•										107
	P	AR	T	D				A	N	IM	P	LE	ME	N	TA:	ri	0	N	OF	M	UL	TI.	PRO	OGI	RAE	HI	NG	; A	NE)	

PROCESS MANAGEMENT FOR A SECURITY KERNEL OPERATING SYSTEM

<u>Chapter</u>																										page
XIII. I	NTR	DU	CT	IO	N		•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•		•	109
XIV. I	IPLE	1EN	TA	TI	01	1	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	112
	DE	EL	OP	ME	NI	C A	L	S	JPI	20	RT	•	•	•		•	•		•			•	•		•	112
	INI	ER	Т	RA	PI	71	С	C	วหา	r R	OL.	L	ΞR						_							114
		Vi	rt	ua	1	P	r	oci	288	50	r	Ta	a b	1 6	٠	14	PT)								114
		Le	٧e	1-	1	S	cl	ne	du i	Li	na			•			•	٠.	•							118
			G	et	wc) I	k																			119
		Vi	rt	ua	1	p	r	o C	259	50	r	Tr	1.5	т †. т	111	ct	io	n	set						-	126
																			•							
			S	ia	na	1	•			_	•			•	•	·	•	Ī	•	•	•	•			Ĭ	130
			S	U 1	p	v	זמ	3 R		•	Ī				Ī	Ī	Ť	Ī		•	•	•	•	Ť	•	131
			T	DT.	_ _	- '		J10	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	133
			Ġ	D.D.	τ	<i>1</i> D	· DI	• । दाद	e 4 D I	• •	•	•		•	•	•	•	•	•	•	•	•	•	•	•	134
			m	TIC	COR.	19	D 1		T 24 T	a																435
	m p i	ספ	TC	دع		1/73	D (1 E 1			•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	133
	LH	111	16		U	A.T.	n (וענ		X	• - h	1	•	•	•	• •	•	•	•	•	•	•	•	•	•	130
		AC	TI.	ve	` E	'E'	00	je:	3S	T	a D	Τ ∈	3	(E	IP.	T)	•	•	•	•	•	•	•	•	•	138
		re	٧e	T-	2	5	CI	160	ıuı	Ll	ng	•	•	•	•	•	•	•	•	•	•	•	•	•	•	141
			T	<u>C</u> _	GI	ST	W	ואנ	ί 	•	•			•	•	•	•	•	•	•	•	•	•	•	•	141
		_	Т	Ç_	21	(15	E	12.	. — t	1 A	ND	L	SK		•	•	•	•	•	•	•	•	•	•	•	138 138 141 141 143 145 146 146
		EV	en	tc	OU	ın	ts	5	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	145
			A	d v	an	C	е	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	145
			A	wa	1.t		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	146
			R	ea	ď		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	146
			-		15 6	-		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	170
	SYS	TE	M	ΙN	II	l!	ΑI	II:	ZAI	ľ	ON	•	•	•	•	•	•	•	•	•	•	٠	•	•	•	147
XV. CON	CLUS	10	N	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	151
	REC	MO	ME	ND	AI	I.	01	IS																		151
	FOI	LO	W	ON	W	10	RF	(•	•	•	•	•	•	•	•	•	•	٠	•	•	•	•	•	•	152
PART	E			IM	ΡI	E	ME	en:	CAT	ï	ON	C	F	S	E	GM	EN:	r	MAN	A	GE	1E	NT	P	OR	Ά
				SE	C	IR	E	Al	RCE	II	VA	L	S	rc	R	AG	E :	SY.	STE	M						
Chapter	•																									page
XAI. IN	TROE	UC'	TI	ON	•		•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	154
XVII. S	EGME	NT	M	AN	ΑG	E	ΜE	e n i	C E	U	NC	TI	01	N S	;	•	•	•	•	•	•	•	•	•	•	155
	SEG	ME	NT	M	A N	IA	GE	ER	•	•	•		, ,	•			•	•	•	•			•			155
		Fu	nc	ti	on	ı											•		•							155
																			•							156
	NON	I-D	IS	CR	ET	I	ON	IAI	RY	S	EC	UE	lI:	ΓY		MO	DUI	LE								160
		OR																								161
																			•							161
																										162

	SU	1MAR	Y	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	166
XVIII.	SEG	1 E N T	M	A N	ΑG	E	ME:	NT	IM	PL	EME	E NT	ΑT	OI	N	•	•	•	•	•	•	٠	167
	IME	PLEM	E N	ΤA	ΤI	01	N	ISS	UE	S													167
		Int																					168
		Str	uc	ŧυ	re	s	a:	s A	rg	un	ent	s	•	•	•	•	•	•	•	•	•	•	170
		Ree	nt	ra	nt	. (Co	de															170
		Pro	ce	ss	S	tı	cu	ctu	re	0	f t	he	M	em	or	У	Ma	na	ge:	r			171
		Per	- P	ro	ce	SS	5	Knc	wn	S	equ	en	t	Ta	b1	ė	•	•	•	•			172
		DBR	H	a n	dl	e																	172
	SEC	MEN	T	MA	N A	.G 1	ER	MC	DU	LE										•			173
		Cre	at	е	a	Se	ear	ner	it	•											•		174
		Cre Del	et	e	a	Se	e a i	nen	t														177
		Mak	e	a	Se	σı	ne:	nt	Kn	o w	n .												178
		Mak	e	a	Se	Q I	ne	nt	Un	kn	OWI	1 (Te	Tm:	in	аt	eì						181
		Swa	ם כ	a	Se	a i	ne:	nt	In			•										•	182
		Swa	D i	a	Se	Q I	ne	nt	Ou	t			•										183
	NON	-DI																					
		Equ																					
		Gre																					
	DIS	TRI																					
		Des																					
		Int																					
	SUN	IMAR																					
			-	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	
XIX. C	ONCLU	SIO	NS	A	ND	I	? 01	LLC	W	ON	WC	RK	•	•	•	•	•	•	•	•	•	•	193
XIX. C		/SIO	I	MP	LE	ME	en:	CAT	'IO	N	0 P	PR	oc	ES	S	MA	NA	GE					
			I	MP	LE	ME	en:	CAT	'IO	N		PR	oc	ES	S	MA	NA	GE					
	r		I	MP	LE	ME	en:	CAT	'IO	N	0 P	PR	oc	ES	S	MA	NA	GE					A
PAR	r P		I:	MP EC	L E U R	ME	EN!	rat RCH	IV	N A L	OP SI	PR OR	OC AG	ES:	5 5 Y .	MA ST	n a E m	GE	ME	NT	P) R	<u>A</u>
P A R	r P		I:	MP EC	L E U R	ME	EN!	rat RCH	IV	N A L	OP SI	PR OR	OC AG	ES:	5 5 Y .	MA ST	n a E m	GE	ME	NT	P	O R	<u>A</u>
PARS	r F E	 UCTI	I S	MP EC	LE UR	MI E	EN S	rat RCH	·io	N A L	OP SI	PR OR	OC AG	ES:	5 5 Y .	MA ST	n a E m	GE	ME	NT	P	O R	<u>page</u>
PAR	r F E	 UCTI	I S	MP EC	LE UR	MI E	EN S	rat RCH	·io	N A L	OP SI	PR OR	OC AG	ES:	5 5 Y .	MA ST	n a E m	GE	ME	NT	P	O R	<u>page</u>
PARS	r F Erodu Mples	LL ICTIO	I: S: ON	MP EC	LE UR	MH E	A)	rat RCH	·	N AL	or si	PR OR	OC AG	ES:	S Y .	MA ST	HA EM	GE!	ME:		P(OR.	<u>page</u> 196
PARS	r F Erodu Mples	JCTIO	I: S: ON AT:	MP EC	LE UR N	HE E	Al	rat RCH	IO	N AL	OF SI	PR OR	OC AG	ES:	5 Y .	MA ST	NA EM	GE	ME:		P (• • • • • • • • • • • • • • • • • • •	196 198
PARS	r F Erodu Mples	JCTICIENT:	I. S. ON AT:	MP EC	LE UR N	nie E	All	FAT RCH	AT Con	N AL	OF SI	PR OR	OC AG	ES:	S SY.	MA ST	NA EM	GE:	ME:	NT .			196 198 198 198
PARS	r F Erodu Mples	JCTIO	II SI ON ATI	MP EC	LE UR N	HIE IS	All	· JES	AT	N AL	OF SI	PR	OCC AG	ES:	S SY.	MA ST	NA EM	GE	ME:	NT ·	P(196 198 198 199
PARS	r F TRODU MPLEM DAI	JCTIC IENT. ABA: Inne	II SI ON SE er ffi	MPPEC IO I Tico	LEUR N NI Cona	IS TI	All	· · · · · · · · · · · · · · · · · · ·	AT on le	N · · · · · · · · · · · · · · · · · · ·	OF SI	PR OR	OC AG	ES:	SY.	MA ST ·	NA EM	GE:	ME:	NT ·	P(OR	196 198 198 199 202 205
PARS	r F TRODU MPLEM DAI	JCTIO	II SE SE er ff: it:	MP EC IO IT io IN	LEUR N NI Ta na	MH E	All	FATRCH	AT on le	N AL	OF ST	PR OR	oc AG	ES:	s sy.	MA ST	NA EM	GE:	ME:	NT ·	P(196 198 198 199 202 205 206
PARS	r F TRODU MPLEM DAI	JCTIO IENT. ABA: Inno Tra: Add: EMP: Phy:	I: S: ON AT: SE er ff: iit:	MP EC IO IT IN IN	LEUR N N NI rac na TE	MH.E.	All	rat RCH JES Collist PTS emp	AT on le	N AL	OF SI	PR OR	OC AG	ES:	S SY.	MA ST aloir 	NA EM	GE:	ME:	NT ·	P(196 198 198 199 202 205 206 207
PARS	T F TRODU	JCTIO IENT: ABA: Inn: Tra: Add: EMP: Phy:	I: S: ON AT: ff: it:	MP EC IO I TCO IN Cal	LE UR N NI Ta C na TE	ME E . IS TI fff or L RFF re	All	rat RCH JES JES Colt Scolt Semp	AT on le	N AL	OP SI	PR OR	OC AG	ES:	s s v · · · · · · · · · · · · · · · · ·	MAA ST	NA EM	GEN at:	ME:	NT ·	P(• • • • • • • • • • • • • • • • • • •	196 198 198 199 202 205 206 207 209
PARS	T F TRODU MPLEM DAI	JCTIC IENT: ABA: Inn: Tra: Add: EMP: Phy: Vir:	I SE SE ff: sictuation	MP EC IO I TCO IN Cal	LEUR N NI a C na E l P S S	MHE.	A) A) CSU A) CSU CSU CSU CSU CSU CSU CSU CS	TATTACH JES Collinit Constant The property of the propert	ATT on le	N	OF SI	PR OR	oc AG	ES:	s sv.	MA ST . aloir 	NA EM	GE!	ME:	NT ·	P(196 198 198 199 202 205 206 207 209 213
PARS	TRODUMPLEM DAI	JCTIO IENT: ABA: Inn: Add: EMP: Phy: Vir:	I SE	MPC O ITCONAL CAL	LEUR N NI TE L PSK	TI find the second seco	All	FAT PAT PAT PAT PAT PAT PAT PAT P	AT on le	N	OP ST	PR OR	oc AG	ES:	s s s s s s s s s s s s s s s s s s s	MA ST . al io ir	NA EM	at:	ME:	NT ·	P(OR	196 198 198 199 202 205 206 207 209 213 215
PARS	TRODUMPLEM DAI	JCTIC IENT: ABA: Inn: Tra: Add: EMP: Phy: Vir:	I SE	MPC O ITCONAL CAL	LEUR N NI TE L PSK	TI find the second seco	All	FAT PAT PAT PAT PAT PAT PAT PAT P	AT on le	N	OP ST	PR OR	oc AG	ES:	s s s s s s s s s s s s s s s s s s s	MA ST . al io ir	NA EM	at:	io:	NT ·	P(OR	196 198 198 199 202 205 206 207 209 213 215
PARS	TRODUMPLEMDAI	ICTICAL ABA: Inna Addi EMP: Physical Pictor ICTICAL ACTOR	II SI ON ATI	MPC IO ITCONALE LAL	LEUR N NITATE 1 PSK	TI ff or LRF Preserved.	All	FAT RCH JES Olitical Pempt L	ATTON le ia	N	OP ST	PROR	OCC AG	ES:	s	MAST.	iz n em	at:	ME:	NT ·	P(OR	196 198 198 199 202 205 207 209 213 215 217

			Si	ı pp	01	ct	₽	r	೦೦	ed	u	re	S	•	•	•		•	•	•	•	•	•	•	•	•	•	221
			Re	ead	١.	•	•	•	•	•		•	•	•		•			•		•	•	•	•	•	•	•	223
			Ti	ick	et	:	•	•	•	•		•		•	•				•	•	•	•	•	•	•	•	•	223
			Av	ai	.t		•	•	•	•		•	•	•	•	•			•	•	•	•	•		•	•	•	224
			Ac	l va	no	e		•	•	•		•	•	•	•				•	•	•		•	•	•	•	•	225
		TRA	F	?IC	: (0	ΝT	R	OL	LE	R	M	0	DU	LΕ	•	•		•	•	•	•	•	•	•	•	•	225
			TO	<u>_</u> G	et	W	or	k	•	•		•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	226
			TO	C_A	Wa	ıi.	t	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	227
			T	_A	dv	7a	nc	e	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	228
			Vi	irt	ua.	11	_₽	r	ee	m p	t	_H	aı	ad.	Le	ľ	•	•	•	•	•	•	•	•	•	•	•	233
			Re	ema	ir	ıi.	ng		Pr	00	е	d u	r	es	•	•	•	•	•	•	•	•	•	•	•	•	•	233
		DIS																										234
			MI	1_F	lea	ıd,	_E	V	en	tc	0	un	t	•	•	•	•	•	•	•	•	•	•	•	•	•	•	235
			MI	1_A	dv	ra:	nc	е	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	235
			MI	1_T	10	:K	et		•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	236
			MI	1_A	11	LO	ca	t	е	•	_	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	236 238
		GAI	E	KE	E	? E.	R	M	DC	UL	E	S	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	238
			Us	ser	_9	a.	te		Mo	du	1	e	•	•	•	•	•		•	•	•	•	•	•	•	•	•	239
			K ∈	ern	el	<u> '</u>	Ga	t	e_	Ke	е	рe	ľ	M	bc	ul	. е		•	•	•	•	•	•	•	•	•	239 242 243
		SUR	IMA	RY	•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	243
XXII	I. (CONC	L	JSI	01	I	•	•	•	•		•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	244
		FOI	LC	W	01	1	WO	R	K	•		•		•		•		•	•	•	•	•	•		•	•	•	245
		_																										
Apper	nal:	<u>x</u>																										page
Α.	EVI	ENT	MI	NA	GE	ER	L	I	ST	IN	G	S	•	•	•	•	•	,	•	•	•	•	•	•	•	•	•	247
D	mp :	APPI		~			0.7			7	_	~ m	.		-													250
В.	TRI	AFFI	C		NI	: R	υL	ابلا	ĽK	L	Ι.	5 T	Τ.	NG:)	•	•	•	•	•	•	•	•	•	•	•	•	258
c.	חדמ	STRI	ים י	7 m T	תי	Mr.	D M	<u> </u>	o V	M		XT 76	~ I	מי	T	TC	m 1	- 3 7/	٠.									28 7
C.	DT:	2 T.W.T	. סי) T. C	עי	il.	e II	O	1 7	i.i.	A	IA H	61	ZX	مد	TO	11	r IA	33		•	•	•	•	•	•	•	201
D.	CAT	re_K	म भ	קסי	· R	τ.	TS	ጥ -	ΓN	<u>ر</u> د																		308
<i>D</i> •	JA.				110	ш.			_ 14	3.5		•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	300
E.	BOO	OTSI	R	P	T.C) A I	DΕ	R	ī.	T S	T	ΤN	G S	3			_				_	_				_		317
2.				-		•		•••	_		-		•		٠	٠	·		•	•	•	•	•	•	•	•	•	• • •
F.	LI	BRAR	Y	FU	NC	T	ΙO	N	L	IS	T	ΙN	G S	5				, ,							•		•	330
G.	INI	NER	TF	RAF	FI	C	C	01	T	RO	L	LE	R	L	S	TI	NG	S		•	•	•	•	•	•	•	•	335
H .	SEC	GMEN	T	MA	NA	G	ER	. 1	LI	ST	I	NG	S	•	•	•	•		•	•	•	•	•	•	•	•	•	364
I.	NON	V-DI	SC	CRE	TI	0	N A	R	ľ	SE	C	UR	I	ľY	L	IS	TI	N	SS		•	•	•	•	•	•	•	385
•		40.5								m =	, .																	205
J.	ME	MORY	Ľ	AN	AG	E	K	L	LS	ΤI	M (GS		•	•	•	•		•	•	•	•	•	•	•	•	•	387
TTCT	0.73	חחח	מים ו	1722	~	3.0																						4.04
LIST	20	ner	L	LLN	C.E	5		•	•	•		•	•	•	•	•	•		•	•	•	•	•	•		•	•	404

INITIAL DISTRIBUTION	LIST	•		•	•	•	•	•	•	•	•	•	•	•	•	40	6
----------------------	------	---	--	---	---	---	---	---	---	---	---	---	---	---	---	----	---

- xxxvii -

LIST OF FIGURES

F=77F	.5												53	146
1.	SASS System Interfaces	•	•	•	•	•	•	•	•	•	•	•	•	A
2.	Extended Machine Layers	•	•	•	•	•	•	•	•	•	•	•	vi	Lii
3.	Internal Kernel Organization	•	•	•	•	•	•	•	•	•	•	•	xi	lii
4.	Multiprocessor Configuration	•	•		•	•	•	•	•	•	•	3	C V i	lii
5.	SASS System	•	•	•	•	•	•	•	•	•	•	•	•	20
6.	System Overview (Dual Host)	•	•	•	•	•	•	•	•	•	•	•	•	22
7.	Known Segment Table (KST) .	•	•	•	•	•	•	•	•	•	•	•	•	31
8.	Active Process Table (APT) .	•	•	•	•	•	•	•	•	•	•	•	•	35
9.	Virtual Processor Table (VPT))	•	•	•	•	•	•	•	•	•	•	•	39
10.	Extended Instruction Set	•	•	•	•	•	•	•	•	•	•	•	•	46
11.	Kernel Databases	•	•	e	•	•	•	•	•	•	•	•	•	47
12.	Memory Management Unit (MMU)	IO	ag	je	•	•	•	•	•	•	•	•	•	50
13.	SASS H/W System Overview	•	•	•	•	•	•	•	•	•	•	•	•	59
14.	Global Active Segment Table	•	•	•	•	•	•	•	•	•	•	•	•	64
15.	Alias Table Creation	•	•	•	•	•	•	•	•	•	•	•	•	67
16.	Local Active Segment Table .	•	•	•	•	•	•	•	•	•	•	•	•	69
17.	Alias Table	•	•	•	•	•	•	•	•	•	•	•	•	70
18.	Memory Management Unit Image	•	•	•	•	•	•	•	•	•	•	•	•	73
19.	Memory Allocation/Deallocation	on	Ma	ιp	•	•	•	•	•	•	•	•	•	75
20.	Memory Manager Mainline Code	•	•	•	•	e	•	•	•	•	•	3	•	77
21.	Create Entry Pseudo-code		•	•	•	•	•	•	•	•	•	•	•	79

22.	perete	Enti	À Pa	seuc	10-0	cod	le	•	•	•	•	•	•	•	•	•	•	•	•	•	•	82
23.	Activa	te Ps	eudo	- co	de	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	86
24.	Deacti	vate	Pseu	ıdo-	cod	le	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	90
25.	Swap_I:	n Pse	udo-	-cod	le	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	94
26.	Swap_0	ut Ps	eudo	o-co	de	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	97
27.	Deacti	vate	A11	Pse	eudo) - C	o đ	le	•	•	•	•	•	•	•	•	•	•	•	•	•	99
28.	Move To	o Glo	bal	Pse	udo) - C	o d	le	•	•	•	•	•	•	•	•	•	•	•	•		100
29.	Move To	o Loc	al E	seu	do-	co	de	•	•	•	•	•	•	•	•	•	•	•	•	•		101
30.	Update	Pseu	do-c	code		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		102
31.	Success	s Cod	es		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	4	104
32.	SASS S	YSTEM	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	4	111
33.	MMU_IM	AGE			•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		113
34.	Virtual	l Pro	cess	sor	Tab	le	<u> </u>	•	•	•	•	•	•	•	•	•	•	•	•	•	1	115
35.	Virtua!	l Pro	cess	sor	Sta	te	s	•	•	•	•	•	•	•	•	•	•	•	•	•	1	117
36.	SWAP_DI	BR .			•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	1	120
37.	Kernel	Stac	k Se	gme	nts	;	•	•	•	•	•	•	•	•	•	•	•	•	•	•	1	124
38.	GETWORE	к .			•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	1	125
39.	Active	Proc	ess	Tab	le	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	1	140
40.	Initial	lized	Sta	ck	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	1	148
41.	Known S	Segme:	nt I	abl	е		•	•	•	•	•	•	•	•	•	•	•	•	•	•	1	159
42.	Memory	Mana	geme	ent	Uni	t	Im	a g	е	•		•	•	•	•	•	•	•	•	•	1	l ó 5
43.	Memory	Mana	ger-	CPU	Ta	bl	е	•	•	•	•	•	•	•	•	•	•	•	•	•	1	166
44.	Initial	l Pro	cess	St	ack		•	•	•	•	•	•	•	•	•	•	•	•	•	•	2	210
45.	Impleme	entat	ion	Mod	ule	S	tr	uc	tu	re		•	•	•	•	•	•	•	•	•	2	19
46.	TC_ADV	ANCE	Algo	rit	hm	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	2	30
47.	Program	n Sta	tus	Are	a							•						•		•	2	241



PART A INTRODUCTION

Chapter I

BACKGROUND

This chapter is an updated excerpt from <u>Implementation of Segment Management for a Secure Archival Storage System</u> by J. T. Wells [20].

O'Connell and Richardson provided the design for a family of secure, distributed, multi-microprocessor operating systems from which the subset, SASS, was later derived [7]. In their work, two of the primary motivations were to provide a system that (1) effectively coordinated the processing power of microprocessors and (2) provided information security.

The basis for emphasis on utilization of microprocessors is not purely that of replacing software with more powerful (and faster) hardware (microprocessors) but is also an economic issue. Software development and computing operations are becoming more and more expensive, putting further pressure on system designers to increasingly utilize people solely for system functions that computers cannot perform in a cost effective manner. Microcomputers, on the other hand, are becoming less and less expensive and are, therefore, increasingly being used for more functions.

The need for information security has been gradually recognized as the uses of computers have expanded. As security

needs for specific computer systems have been recognized, attempts have been made to modify the existing systems to provide the desired security. The results have been systems that could not be certified as secure and/or which have failed to resist penetration efforts, i.e. systems which, in effect, did not provide adequate information security. It has become clear that, in order to be certifiably secure, a computer system must have security designed in from first principles [10,11]. Such is the case with SASS. Information security was and continues to be a chief design feature. Integral to the design goal of information security were two related goals. One of these goals was to provide multilevel controlled access to a consolidated warehouse of data for a network of multiple host computers. The other key goal was to provide for controlled sharing among the computer hosts.

A brief background of prior work relative to SASS follows. O'Connell and Richardson originated the design of a secure family of operating systems. Their design provided two basic parts for their system — the supervisor (to provide operating system services) and the kernel (to provide for physical resource management). The design of the SASS supervisor was completed by Parks [9]. No implementation or further design effort on the supervior has followed, to date. The initial design of the kernel was completed by Coleman [2]. That design described the kernel in terms of seven modules:

- 1. Gate Keeper Module -- provided for ring-crossing mechanism and thus isolation of the kernel.
- Segment Manager Module -- provided for management of segmented virtual memory.
- 3. Traffic Controller Module -- multiplexed processes onto virtual processors and supports the inter- process communication primitives Block and Wakeup.
- 4. Non-Discretionary Security Module -- mediated non-discretionary security access attempts.
- 5. Inner Traffic Controller Module -- multiplexed virtual processors onto real processors and provided the Kernel synchronization primitives Signal and Wait.
- 6. Memory Manager Module -- managed main memory and secondary storage.
- Input-Output Manager -- managed the moving of information to external devices outside the boundaries of the SASS.

Refinement of the kernel design and partial implementation was completed by Gary and Moore [5] in conjunction with Reitz [12]. The resultant description of the kernel as a result of their work was:

- 1. Gate Keeper Module
- 2. Segment Manager Module
- 3. Event Manager Module -- worked with the Traffic Controller to manage the event data associated with the IPC mechanism of eventcounts and sequencers.
- 4. Non-Discretionary Security Module
- 5. Traffic Controller Module -- replaced Block and Wakeup with Advance and Await (to implement Supervisor IPC mechanism of eventcounts and sequencers).
- 6. Memory Manager Module
- 7. Inner Traffic Controller Module

Reitz implemented the Traffic Controller Module and Inner Traffic Controller Module. Gary and Moore completed a detailed design of the Memory Manager, originated the Memory Manager code (written predominantly in PLZ/SYS), selected a thread of the code, hand compiled it into PLZ/ASM and ran it on the Z8000 developmental module. Wells provided the implementation of the Segment Manager and Non-Discretionary Security Modules as well as partial implementation of Distributed Memory Manager functions. Strickler refined and implemented the process management functions for the SASS (written in PLZ/ASM).

Chapter II

BASIC CONCEPTS/DEFINITIONS

This chapter is an excerpt from <u>Implementation of</u>
Process <u>Management</u> for a <u>Secure Archival Storage</u>
System by A. R. Strickler [19]. Minor changes
have been made for integration into report.

This section provides an overview of several concepts essential to the SASS design. Readers familiar with SASS or with secure operating system principles may wish to skip to the next section.

A. PROCESS

The notion of a process has been viewed in many ways in computer science literature. Organick [8] defines a process as a set of related procedures and data undergoing execution and manipulation, respectively, by one of possibly several processors of a computer. Madnick and Donovan [6] view a process as the locus of points of a processor executing a Reed [10] describes a collection of programs. process as the sequence of actions taken by some processor. In other words, it is the past, present, and future "history" of the In the SASS design, a process is states of the processor. viewed as a logical entity entirely characterized by an address space and an execution point. A process address space consists of the set of all memory locations accessible by the process during its execution. This may be viewed as a set of procedures and data related to the process. The execution point is defined by the state of the processor at any given instant of process execution.

As a logical entity, a process may have logical attributes associated with it, such as a security access class, a unique identifier, and an execution state. This notion of logical attributes should not be confused with the more typical notion of physical attributes, such as location in memory, page size, etc. In SASS, a process is given a security access class, at the time of its creation, to specify what authorization it possesses in terms of information access (to be discussed in the next section). It is also given a unique identifier that provides for its identification by the system and is utilized for interaction among processes. A process may exist in one of three execution states: 1) running, 2) ready, and 3) blocked. In order to execute, a process must be mapped onto (bound to) a physical processor in the system. Such a process is said to be in the "running" state. A process that is not mapped onto a physical processor, but is otherwise ready to execute, is in the "ready" state. A process in the "blocked" state is waiting for some event to occur in the system and cannot continue execution until the event occurs. At that time, the process is placed into the ready state.

B. INFORMATION SECURITY

There is an ever increasing demand for computer systems that can provide controlled access to the data it stores. In this thesis, "information security" is defined as the process of controlling access to information based upon proper authorization. The critical need for information security should be clear. Banks and other commercial enterprises risk the theft or loss of funds. Insurance and credit companies are bound by law to protect the private or otherwise personal information they maintain on their cus-Universities and scientific institutions must prevent the unauthorized use of their often over-burdened sys-The Department of Defense and other government tems. agencies must face the very real possibility that classified information is being compromised or that weapon systems are being tampered with. In fact, security related problems can be found at virtually every level of computer usage.

The security of computer systems processing sensitive information can be achieved by two means: external security controls and internal security controls. In the first case, security is achieved by encapsulating the computer and all its trusted users within a single security perimeter establishe by physical means (e.g., armed guards, fences, etc.) This means of security is often undesirable due to its added cost of implementation, the inherent risk of error-prone manual procedures, and the problem of trustworthy but error-

prone users. Also, since all security controls are external to the computer system, the computer is incapable of securely handling data at differing security levels or users with differing degrees of authorization. This restriction greatly limits the utility of modern computers. Internal security controls rely upon the computer system to internally dismultiple levels between ο£ classification and user authorization. This is clearly a more desirable and flexible approach to information security. This does not mean, however, that external security is not needed. The optimal approach would be to utilize internal security controls to maintain information security and external security controls to provide physical protection of our system against sabotage, theft, or destruction. The primary concern of this thesis is information security and will therefore center its discussion on the achievement of information security through implementation of the security kernel concept.

One might argue that a "totally secure" computer system is one that allows no access to its classified or otherwise sensitive information. Such a system would not be of much value to its users. Therefore, when we say that a system provides information security, it is only secure with respect to some specific external security policy established by laws, directives, or regulations. There are two distinct aspects of security policy: non-discretionary and discre-

tionary. Each user (subject) of the system is given a label denoting what classification or level of access the user is authorized. Likewise, all information or segments (objects) within the system are labelled with their classification or level of sensitivity. The non-discretionary security mechanism is responsible for comparing the authorization of a subject with the classification of an object and determining what access, if any, should be granted. The DOD security classification system provides an example of the non-discretionary security policy and is the policy implemented in The discretionary security policy is a refinement of the non-discretionary policy. As such, it adds a higher degree of restriction by allowing a subject to specify or restrict who may have access to his files. It must be emphasized that the discretionary policy is contained within the non-discretionary policy and in no way undermines or substitutes for it. This prevents a subject from granting access that would violate the non-discretionary policy. An example of discretionary security is provided by the DOD "need to know" policy. In SASS, the discretionary policy is implemented within the supervisor [9] by means of an Access Control List (ACL). There is an ACL maintained for every file in the system, which provides a list of all users authorized access to that file. Every attempt by a user to access a file is first checked against the ACL and then checked against the non-discretionary security policy. The "least"

or "most restrictive" access found in these checks is then granted to the user.

The relationship between the labels associated with the subject's access class (sac) and the object's access class (oac) is defined by a lattice model of secure information flow [12] as follows ("!" denotes "no relationship"):

- 1. sac = oac, read and write access permitted
- sac > oac, read access permitted
- 3. sac < oac, write access permitted
- 4. sac | oac, no access permitted

In order to understand how these access levels are determined, it is necessary to gain an awareness of and consideration for several basic security properties.

The "Simple Security Property" deals with "read" access. It states that a subject may have read access only to those object's whose classification is less than or equal to the classification of the subject. This prevents a subject from reading any object possessing a classification higher than his own.

The "Confinement Property" (also known as "*-property") governs "write" access. It states that a user may be granted write access only to those objects whose classification is greater than or equal to the classification of the subject. This prevents a user from writing information of a higher classification (e.g., Secret) into a file of a lower classification (e.g., Unclassified). It is noted that while

this property allows a user to write into a file possessing a classification higher than his own, it does not allow him access to any of the data in that file. The SASS design does not allow a user to "write up" to higher classified files. Therefore, in SASS, "sac < oac" denotes "no access permitted."

The "Compatibility Property" deals with the creation of objects in a hierarchical structure. In SASS, objects (segments) are hierarchically organized in a tree structure. This structure consists of nodes with a root node from which the tree eminates. The Compatibility Property states that the classification of objects must be non-decreasing as we move down the hierarchical structure. This prevents a parent node from creating a child node of a lower classification.

Several prerequisites must be met in order to insure that the security kernel design provides a secure environment. Firstly, every attempt to access data must invoke the Kernel. In addition, the Kernel must be isolated and tamperproof. Finally, the Kernel design must be verifiable. This implies that the mathematical model, upon which the Kernel is based, must be proved secure and that the Kernel is shown is to correctly implement this model.

C. SEGMENTATION

Segmentation is a key element of a security Kernel based system. A segment can be defined as a logical grouping of information, such as a procedure, file or data area [6]. Therefore, we can redefine a process' address space as the collection of all segments addressable by that process. Segmentation is the technique applied to effect management of those segments within an address space. In a segmented environment, all references within an address space require two components: 1) a segment specifier (number) and 2) the location (offset) within the segment.

A segment may have several logical and physical attributes associated with it. The logical attributes may include the segment's classification, size, or permissable access (read, write, or execute). These logical attributes
allow a segment to nicely fit the definition of an object
within the security kernel concept, and thus provide a means
for the enforcement of information security. A segment's
physical attributes include the current location of the segment, whether or not the segment resides in main memory or
secondary storage, and where the segment's attributes are
maintained by a segment descriptor. The segment descriptors
for each segment in a process' address space are contained
within a Descriptor Segment (viz., the MMU Image in SASS) to
facilitate the memory management of that address space.

Segmentation supports information sharing by allowing a single segment to exist in the address spaces of multiple processes. This allows us to forego the maintenance of multiple copies of the same segment and eliminates the possibility of conflicting data. Controlled access to a segment is also enforced, since each process can have different attributes (read/write) specified in its segment descriptor. In the implementation of SASS, any segment which is shared, but has "read only" access by every process sharing it, is placed in the processor local memory supporting each of these processes rather than in the global memory. This implies the maintenance of multiple copies of some shared seg-It is noted that the problem of "conflicting data" ments. is avoided since this only applies to read only segments. This apparent waste of memory and nonuse of existing sharing facilities is justified by a design decision to provide maximum reduction of bus contention among processors accessing global memory. This reduction in bus contention is considered to be of more importance than the saving of memory space provided by single copy sharing of read only segments. This decision is also well supported by the occurrence of decreasing memory costs, which we have experienced in terms of high speed bus costs.

D. PROTECTION DOMAINS

The requirement for isolating the Kernel from the remainder of the system is achieved by dividing the address space of each process into a set of hierarchical domains or protection rings [18]. O'Connell and Richardson [7] defined three domains in the family of secure operating systems: the user, the supervisor, and the kernel. Only two domains are actually necessary in the SASS design since it does not provide extended user applications. The Kernel resides in the inner or most privileged domain and has access to all segments in an address space. System wide data bases are also maintained within the Kernel domain to insure their accessibility is only through the Kernel. The Supervisor exists in the outer or least privileged domain where its access to data or segments within an address space is restricted.

While protection domains may be created through either hardware or software mechanisms, a hardware implementation provides much greater efficiency. Current microprocessor technology only provides for the implementation of two domains. This two domain restriction does not support O'Connell and Richardson's complete family design, but it is sufficient to allow hardware implementation of the ring structure required by the SASS subset.

E. ABSTRACTION

Dijkstra [4] has shown that the notion of abstraction can be used to reduce the complexity of a problem by applying a general solution to a number of specific cases. A structure of increasing levels of abstraction provides a powerful tool for the design of complex systems and generally leads to a better design with greater clarity and fewer errors.

Each level of abstraction creates a virtual hierarchical machine [6] which provides a set of "extended instructions" to the system. A virtual machine cannot make calls to another virtual machine at a higher level of abstraction and in fact is unaware of its existence. This implies that a level of abstraction is independent of any higher levels. This independence provides for a loop-free design. Additionally, a higher level may only make use of the resources of a lower level by applying the extended instruction set of the lower level virtual machine. Therefore, once a level of abstraction is created, any higher level is only interested in the extended instruction set it provides and is not concerned with the details of its implementation. In SASS. once a level of abstraction is created for the physical resources of the system, these resources become "virtualized" making the higher levels of the design independent of the physical configuration of the system.

PART B SECURE ARCHIVAL STORAGE SYSTEM DESIGN

This section is an excerpt from <u>Implementation of Process</u>

<u>Management for a Secure Archival Storage System</u> by A. R.

Strickler [19]. Minor changes have been made for integration into this report.

Chapter III

BASIC SASS OVERVIEW

The purpose of the Secure Archival Storage System is to provide a secure "data warehouse" or information pool which can be accessed and shared by a variable set of host computer systems possessing differing security classifications. The primary goals of the SASS design are to provide information security and controlled sharing of data among system users.

Figure 5 provides an example of a possible SASS usage. The system is used exclusively for managing an archival storage system and does not provide any programming services to Thus the users of the SASS may only create, its users. store, retrieve, or modify files within the SASS. The host computers are hardwired to the system via the I/O ports of the Z8001 with each connection having a fixed security classification. Each host must have a separate connection for each security level it wishes to work on (It is important to note that Figure 5 only represents the logical interfacing of the system. Specifically, the actual connection with the host system must be interfaced with the Kernel as the I/O instructions for the port are privileged). In our example, Host #1 can create and modify only Top Secret files, but it

can read files which are Top Secret, Secret, Confidential, or Unclassified. Likewise, Host #2 can create or modify secret files, using its secret connection or confidential files, using its confidential connection. Host #2 cannot create or modify Top Secret or Unclassified files.

In order to provide information security and controlled sharing of files, the SASS operates in two domains: (1) the Supervisor domain and (2) the Kernel domain. The SASS achieves this desired environment through a distributed operating system design which consists of two primary modules: the Supervisor and the Security Kernel. Each host system connected to the SASS has associated with it two processes within the SASS which perform the data transfer and file management on behalf of that host. The host computer communicates directly with its own I/O process and File Manager process within the SASS.

We can use our notion of abstraction to present a system overview of the SASS. The SASS consists of four primary levels of abstraction:

Level 3-The Host Computer Systems

Level 2-The Supervisor

Level 1-The Security Kernel

Level 0-The SASS Hardware

A pictorial representation of this abstract system overview is presented in Figure 6. This representation is limited to a dual host system for clarity and space restrictions. Note

 Host1	Host2	Host3	 Host4
TI	SI C	•	υį
0 9	el o	0	n C
	ri f		11
Si	ei i	i	a į
e	t! d	l dl	sl
C	l e	e l	s
E1	l ni	n t	i(f(
ti	i	i	i
i	i a	al	el
1	1 1	11	d1
Į,	1	1	1
1	11		1
	[]		
	1 1	1 1.	
SASS	1		
1	Sup	ervisor	i
1			1
1	1	ernel	
1	1	1	
1	1	4	1
1			1
1 1	Main	Second	dary
i	Memory	Store	age
1			
1			ļ

Figure 5: SASS System

that the Gate Keeper module is in actuality the logical boundary between levels one and two and as such will be described separately.

Level 3, the host computer systems, of SASS has already been addressed. It should be noted that the SASS design makes no assumptions about the host computer systems. Therefore each host may be of a different type or size (i.e. micro, mini, or maxi-computer system). Furthermore, the necessary physical security of the host systems and their respective data links with the SASS is assumed.

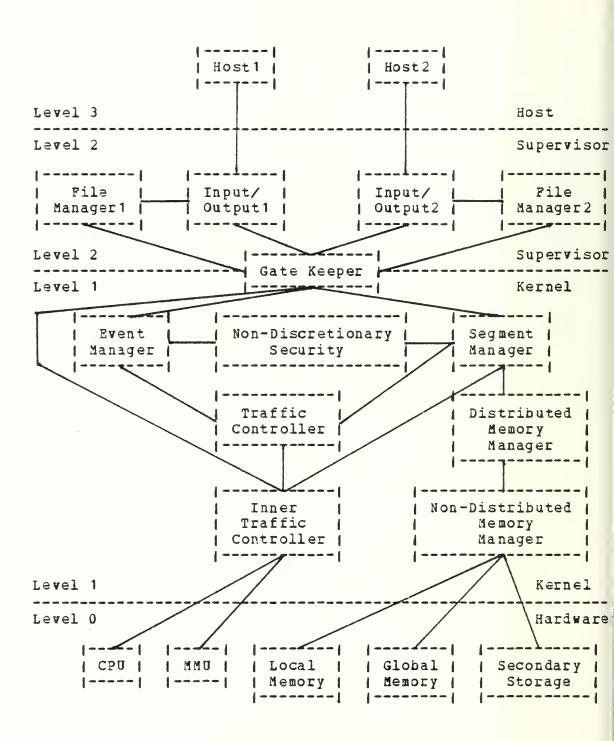


Figure 6: System Overview (Dual Host)

Chapter IV

SUPERVISOR

Level 2 of the SASS system is composed of the Supervisor domain. As already stated, the SASS consists of two domains. The actual implementation of these domains was greatly simplified since the Z8001 microprocessor provides two modes of execution. The system mode, with which the Kernel was implemented, provides access to all machine instructions and all segments within the system. The normal mode, with which the Supervisor was implemented, only provides access to a limited subset of machine instructions and segments within the system. Therefore, the Supervisor operates in an outer or less privileged domain than the Kernel.

The purpose of the Supervisor is to manage the data link between the host computer systems and the SASS by means of Input/Output control, and to create and manage the file hierarchy of each host within the SASS. These functions are accomplished via an Input/Output (I/O) process and a File Manager (FM) process within the Supervisor. A separate FM and I/O process are created and dedicated to each host at the time of system initialization.

A. FILE MANAGER PROCESS

The PM process directs the interaction between the host computer systems and the SASS. It interprets all commands received from the Host computer and performs the necessary action upon them through appropriate calls to the Kernel. The primary functions of the PM process are the management of the Host's virtual file system and the enforcement of the discretionary security policy.

The virtual file system of the Host is viewed as a hierarchy of files which are implemented in a tree structure. The five basic actions which may be initiated upon a file at this level are: 1) to create a file, 2) to delete a file, 3) to read a file, 4) to store a file, and 5) to modify a file. The FM process utilizes a FM Known Segment Table (FM_KST) as the primary database to aid in this management.

The FM process maintains an Access Control List (ACL) through which it enforces the discretionary security in SASS. The FM process initializes an ACL for every file in its Host's file system. The ACL is merely a list of all users that are authorized to access that file. The ACL is checked upon every attempt to access a file to determine its authorization. The user (host computer) directs the FM process as to what entries or deletions should be made in the ACL, and as such, specifies who he wishes to have access to his file. As noted earlier, discretionary security is a refinement to the Non-Discretionary Security Policy and there-

fore can only be utilized to add further access restrictions to those provided by the Non-Discretionary Security. This prevents a user from granting access to a file to someone who otherwise would not be authorized access.

B. INPUT/OUTPUT PROCESS

The I/O process is responsible for managing the input and output of all data between the host computer systems and the SASS. The I/O process is subservient to the FM process and receives all of its commands from it. Data is transferred between the SASS and Host Computer systems in fixed size "packets". These packets are broken up into three basic types: 1) a synchronization packet, 2) a command packet, and 3) a data packet. In order to insure reliable transmission and receipt of packets between the Host computer and the SASS, there must exist a protocol between them. Parks [9] provides a more detailed description of these packets, and a possible multi-packet protocol.

Chapter V

GATE KEEPER

The primary objective of the gate keeper is to isolate the Kernel and make it tamperproof. This goal is accomplished by reason of a software ring crossing mechanism provided by the gate keeper. In terms of SASS, this notion of "ring-crossing" is merely the transition from the Supervisor domain to the Kernel domain. As noted earlier, the gate keeper establishes the logical boundary between the Supervisor and the Kernel, and as a matter of course, it provides a single software entry point (enforced by hardware) into the Kernel. Therefore, any call to the Kernel must first pass through the gate keeper.

The gate keeper acts as a trap handler. Once it is invoked by a user (Supervisor) process, the hardware preempt interrupts are masked, and the user process' registers and stack pointer are saved (within the kernel domain). It then takes the argument list provided by the caller and validates these passed parameters to insure their correctness. To aid in the validation of these parameters, the gate keeper utilizes the Parameter Table as a database. The Parameter table contains all of the permitted functions provided by the Kernel. These relate directly to the extended instruction

set (viz., Supervisor calls) provided by the Kernel (these extended instructions will be described in the next section). If an invalid call is encountered by the gate keeper, an error code is returned, and the Kernel is not invoked. If a valid call is encountered by the gate keeper, the arguments and control are passed to the appropriate Kernel module.

Once the Kernel has completed its action on the user request, it passes the necessary parameters and control back to the gate keeper. At this point, the gate keeper determines if any software virtual preempt interrupts have occurred. If they have, then the virtual preempt handler is invoked vice the Kernel being exited (virtual interrupt structure is discussed by Strickler [19]. Correspondingly, if a software virtual preempt has not occurred, then the return arguments are passed to the user process. The user process' registers and stack pointer (viz., its execution point) are restored and control returned to the Supervisor domain. A detailed description of the Gate Keeper interface and implementation is provided by Strickler [19].

Chapter VI

DISTRIBUTED KERNEL

Level 1 of our abstract view of SASS consists of two components: the distributed Kernel and the non-distributed Kernel. These two elements comprise the Security Kernel of the SASS. The Security Kernel has two primary objectives:

1) the management of the system's hardware resources, and 2) the enforcement of the non-discretionary security policy. It executes in the most privileged domain (viz., the system mode of the Z8001) and has access to all machine instructions. The following section will provide a brief description of the distributed Kernel, its components, and the extended instruction set it provides. A discussion of the non-distributed Kernel will be given in the next section.

The distributed Kernel consists of those Kernel modules whose segments are contained (distributed) in the address space of every user (Supervisor) process. Thus, in effect, the distributed Kernel is shared by all user processes in the SASS. The distributed Kernel is composed of the Segment Manager, the Event Manager, the Non-Discretionary Security Module, the Traffic Controller, the Inner Traffic Controller, and the Distributed Memory Manager Module. The Segment Manager and the Event Manager are the only "user visible"

modules in the distributed Kernel. In other words, the set of extended instructions available to user processes invokes either the Segment Manager or the Event Manager.

A. SEGMENT MANAGER

The objective of the Segment Manager is the management of a process' segmented virtual storage. The Segment Manager is invoked by calls from the Supervisor domain via the gate keeper. Calls to the Segment Manager are made by means of six extended instructions provided by the segment manag-These extended instructions (viz., entry points) are: er. 1) CREATE_SEGMENT, 2) DELETE_SEGMENT, 3) MAKE_KNOWN, 4) TERMINATE, 5) SM_SWAP_IN, and 6) SM_SWAP_OUT. The extended instructions CREATE_SEGMENT and DELETE_SEGMENT add and remove segments from the SASS. MAKE KNOWN and TERMINATE add and remove segments from the address space of a process. Pinally, SM_SWAP_IN and SM_SWAP_OUT move segments from secondary storage to main storage and vice versa.

The primary database utilized by the Segment Manager is the Known Segment Table (KST). A representation of the structure of the KST is provided in Figure 7. The KST is a process local database that contains an entry for every segment in the address space of that process. The KST is indexed by segment number with each record of the KST containing descriptive information for a particular segment. The KST provides a mapping mechanism by which the segment number

of a particular segment can be converted into a unique handle for use by the Memory Manager. The Memory Manager will be discussed in the next chapter.

•	Segment #							
	MM Handle						Entry	
i				 			 	
A 1								
	(

Figure 7: Known Segment Table (KST)

B. EVENT MANAGER

The purpose of the Event Manager is the management of event data which is associated with interprocess communications within the SASS. This event data is implemented by means of eventcounts (a synchronization primitive discussed by Reed [11]). The Event Manager is invoked, via the Gate Keeper, by user processes residing in the Supervisor domain. There are two eventcounts associated with every segment existing in the Supervisor domain. These eventcounts (viz., Instance 1 and Instance 2) are maintained in a database residing in the Memory Manager. The Event Manager provides its management functions through its extended instruction TICKET, ADVANCE, and AWAIT, and in conjunction set READ, with the extended instructions TC_ADVANCE and TC_AWAIT provided by the Traffic Controller (to be discussed next). These extended instructions are based on the mechanism of eventcounts and sequencers [11]. The Event Manager verifies the access permission of every interprocess communication request through the Non-Discretionary Security Module. extended instruction READ provides the current value of the eventcount requested by the caller. IICKET provides a complete time ordering of possibly concurrent events through the mechanism of sequencers. The Event Manager will be discussed in more detail by Strickler [19].

C. NON-DISCRETIONARY SECURITY MODULE

The purpose of the Non-Discretionary Security Module (NDS) is the enforcement of the non-discretionary security policy of the SASS. While the current implementation of SASS represents the Department of Defense security policy, any security policy which may be represented through a lattice structure [3] may also be implemented. The NDS is invoked via its extended instruction set: CLASS EQ and CLASS GE. The NDS is passed two classifications which it compares and then analyzes their relationship. CLASS_EQ will return a true value to the calling procedure only if the two classifications passed were equal. The CLASS GE instruction will return true if a given classification is analyzed to be either greater than or equal to another given classification. The NDS does not utilize a data base as it works only with the parameters it is passed.

D. TRAFFIC CONTROLLER

The task of processor scheduling is performed by the traffic controller. Saltzer [14] defines traffic controller as the processor multiplexing and control communication section of an operating system. The current SASS design utilizes Reed's [10] notion of a two level traffic controller, consisting of: 1) a Traffic Controller (TC) and 2) an Inner Traffic Controller (ITC).

The primary function of the Traffic Controller is the scheduling (binding) of user processes onto virtual proces-A virtual processor (VP) is an abstract data structure that simulates a physical processor through the preservation of an executing process attributes (viz., the execution point and address space). Multiple VP's may exist for every physical processor in the system. Two VP's are permanently bound to Kernel processes (viz., Memory Manager and Idle) and as such are not in contention for process scheduling. These processes and their corresponding virtual processors are invisible to the TC. The remaining virtual processors are either idle or are temporarily bound to user processes as scheduled by the TC. The database utilized by the TC in process scheduling is the Active Process Table (APT). Figure 8 provides the structure of the APT.

The APT is a system-wide Kernel database containing an entry for every user process in the system. Since the current SASS design does not provide for dynamic process creation/deletion, a user process is active for the life of the system. Therefore, the size of the APT is fixed at the time of system generation. The APT is logically composed of three parts: 1) an APT header, 2) the main body of the APT, and 3) a VP table. The APT header includes: 1) a Lock to provide for a mutual exclusion mechanism, 2) a Running List indexed by VP ID to identify the current process running on each VP, 3) a Ready List, which points to the linked list of

								1		- 1		
			Lock									
		į	Running List		A	APT Entry #						
			VP ID						1			
		1	l 4						1	1		
			Ready List Head		- :	 APT Entry #1			HEADER			
			1			1				į		
			Log_CPU_No							1		
		V						1				
	Blocked List Head						i		i			
1	-APT H	enti	: у #									
1					- (11	Awa.	 ited	Event
i	1 30		ם כ	\ \	 S Prior	-i + 17	15+2+0	, λffi_				
i				Class		- 1 c y		nity				ce
1	 	 		 	-		 	 	1 1		 	Count
i					. j			 !	i i		<u> </u>	i i
1	1									,	1	
1	 	 	!		-1			 	1			
A					-			 				1
					-				i i		i	ii
		Lo	og_CI	-0K_U9)	>					
	INR_OF_VP'S!						-	-	- - 	{	TC	
		1					· i	-	-		V P	TO.

| TABLE

Figure 8: Active Process Table (APT)

processes which are ready for scheduling, and 4) a Blocked List, which points to the linked list of processes which are in the blocked state awaiting the occurrence of some event.

A design decision was made to incorporate a single list of blocked processes instead of the more traditional notion of separate lists per eventcount because of its simplicity and its ease of implementation. This decision does not appreciably affect system performance or efficiency as the "blocked" list will never be very long. The VP table is indexed by logical CPU number and specifies the number of VP's associated with the logical CPU and its first VP in the Running List. The logical CPU number, obtained during system initialization, provides a simple means of uniquely identifying each physical CPU in the system. The main body of the APT contains the user process data required for its efficient control and scheduling. NEXT_AP provides the linked list threading mechanism for process entries. The DBR entry is a handle identifying the process' Descriptor Segment which is employed in process switching and memory manage-The ACCESS_CLASS entry provides every process with a security label that is utilized by the Event Manager and the Segment Manager in the enforcement of the Non-Discretionary Security Policy. The PRIORITY and STATE entries are the primary data used by the Traffic Controller to effect process scheduling. AFFINITY identifies the logical CPU which is associated with the process. VP ID is utilized to identify the virtual processor that is currently bound to the process. Finally, the EVENTCOUNT entries are utilized by the TC to manage processes which are blocked and awaiting the occurrence of some event. HANDLE identifies the segment associated with the event, INSTANCE specifies the event, and COUNT determines which occurrence of the event is needed.

The Traffic Controller determines the scheduling order by process priority. Every process is assigned a priority at the time of its creation. Once scheduled, a process will run on its VP until it either blocks itself or it is preempted by a higher priority process. To insure that the TC will always have a process available for scheduling, there logically exists an "idle" process for every VP visible to the TC. These "idle" processes exist at the lowest process priority and, consequently, are scheduled only if there exists no useful work to be performed.

The Traffic Controller is invoked by the occurrence of a virtual preempt interrupt or through its extended instruction set: ADVANCE, AWAIT, PROCESS_CLASS, and GET_DBR_NUMBER. ADVANCE and AWAIT are used to implement the IPC mechanism envoked by the Supervisor. PROCESS_CLASS and GET_DBR_NUMBER are called by the Segment Manager to ascertain the security label and DBR handle, respectively, of a named process. A more detailed discussion of the TC is provided by Strickler [19].

E. INNER TRAFFIC CONTROLLER

The Inner Traffic Controller is the second part of our two-level traffic controller. Basically, the ITC performs two functions. It multiplexes virtual processors onto the actual physical processors, and it provides the primitives for which inter-VP communication within the Kernel is implemented. A design choice was made to provide each physical processor in the system with a small fixed set of virtual processors. Two of these VP's are permanently bound to the Kernel processes. The Memory Manager is bound to the highest priority VP. Conversely, the Idle Process is bound to the lowest priority VP and, as a result, will only be scheduled if there exists no useful work for the CPU to perform. The primary database utilized by the ITC is the Virtual Processor Table (VPT). Figure 9 illustrates the VPT.

The VPT is a system wide Kernel database containing entries for every CPU in the system. The VPT is logically composed of four parts: 1) a header, 2) a VP data table, 3) a message table, and 4) an external VP list. The header includes a LOCK (spin lock) that provides a mutual exclusion mechanism for table access, a RUNNING LIST (indexed by logical CPU #) that identifies the VP currently running on the corresponding physical CPU, a READY LIST (indexed by logical CPU #) which points to the linked list of VP's which are in the "ready" state and awaiting scheduling on that CPU, and a PREE LIST which points to the linked list of unused entries

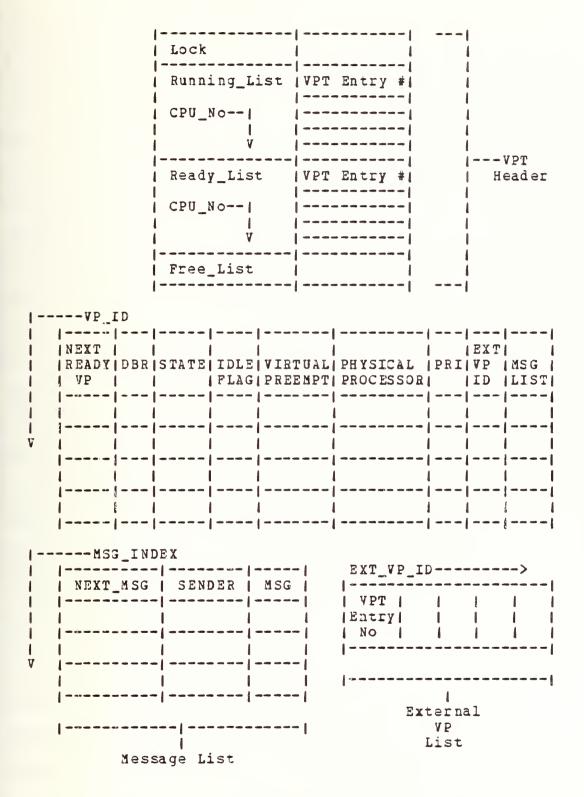


Figure 9: Virtual Processor Table (VPT)

in the message table. The VP data table contains the descriptive data required by the ITC to effectively manage the virtual processors. The DBR entry points within the MMU Inage to the descriptor segment for the process currently running on the VP. PRI (Priority), STATE, IDLE_FLAG, and PREEMPT are the primary data used by the ITC for VP scheduling. PREEMPT indicates whether or not a virtual preempt is pending for the VP. The IDLE_FLAG is set whenever the TC has bound an "idle" process to the VP. Normally, a VP with the IDLE_FLAG set will not be scheduled by the ITC as it has no useful work to perform. In fact, such a VP will only be scheduled if the PREEMPT flag is set. This scheduling will allow the VP to be given (bound) to another process. PHYSICAL PROCESSOR contains an entry from the Processor Data Segment (PRDS) that identifies the physical processor that the VP is executing on. EXT_VP_ID is the identifier by which the VP is known by the Traffic Controller. A design choice was made to have the EXT_VP_ID equate to an offset into the External VP List. The External VP List specifies the actual VP ID (viz., VPT entry number) for each external VP identifier. This precluded the necessity for run time calculation of offsets for the EXT_VP_ID. NEXT_READY_VP provides the threading mechanism for the "Ready" linked list, and MSG_LIST points to the first entry in the Message Table containing a message for that VP. The Message Table provides storage for the messages generated in the course of

Inter-Virtual Processor communications. MSG contains the actual communication being passed, while SENDER identifies the VP which initiated the communication. NEXT_MSG provides a threading mechanism for multiple messages pending for a single VP.

The ITC is invoked by means of its extended instruction set: WAIT, SIGNAL, SWAP_VDBR, IDLE, SET_PREEMPT, and RUNNING_VP. WAIT and SIGNAL are the primitives employed in implementing the Inter-VP communication. SWAP_VDBR, IDLE, SET_PREEMPT, and RUNNING_VP are all invoked by the Traffic Controller. SWAP_VDBR provides the means by which a user process is temporarily bound to a virtual processor. IDLE binds the "Idle" process to a VP (the implication of this instruction will be discussed later). SET_PREEMPT provides the means of indicating that a virtual preempt interrupt is pending on a VP (specified by the TC) by setting the PREEMPT flag for that VP in the VPT. RUNNING_VP provides the TC with the external VP ID of the virtual processor currently running on the physical processor.

F. DISTRIBUTED MEMORY MANAGER

The Distributed Memory Manager provides an interface structure between the Segment Manager and the Memory Manager Process. This interfacing is necessitated by the fact that the Memory Manager Process does not reside in the Distributed Kernel and consequently is not included in the user pro-

cess address space. The primary functions performed in this module are the establishment of Inter-VP Communication between the VP bound to its user process and the VP permanently bound to the Memory Manager Process, the manipulation of event data, and the dynamic allocation of available memory. The Distributed Memory Manager Module is invoked by the Manager through its extended instruction set: Segment MM CREATE ENTRY. MM DELETE ENTRY, MM ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. These extended instructions are utilized on a one to one basis by the extended instruction set of the Segment Manager (e.q., SM_SWAP_IN utilizes (calls) MM SWAP_IN). Wells [20] provides a more detailed description of this portion of the Distributed Memory Manager and the extended instruction set associated with it.

The Distributed Memory Manager is also invoked through its remaining extended instructions: MM_READ_EVENTCOUNT, MM_TICKET, MM_ADVANCE, and MM_ALLOCATE. These Distributed Memory Manager functions are discussed in detail by Strickler [19].

Chapter VII

NON-DISTRIBUTED KERNEL

The Non-Distributed Kernel is the second element residing in Level 1 of our abstract system view of the SASS. The sole component of the Non-Distributed Kernel is the Memory Manager Process.

A. MEMORY MANAGER PROCESS

The primary purpose of the Memory Manager Process is the management of all memory resources within the SASS. These include the local and global main memories, as well as the hard-disk based secondary storage. A dedicated Memory Manager Process exists for every CPU in the system. Each CPU possesses a local memory where process local segments and shared, non-writeable segments are stored. There is also a global memory, to which every CPU has access, where the shared, writeable segments are stored. It is necessary to store these shared, writeable segments in the global memory to ensure that a current copy exists for every access.

The Memory Manager Process is tasked by other processes within the Kernel domain (via Signal and Wait) to perform memory management functions. These basic functions include the allocation/deallocation of local and global memory and

of secondary storage, and the transfer of segments between the local and global memory and between secondary storage and the main memories. The extended instruction set provided by the Memory Manager Process includes: CREATE_ENTRY, DELETE_ENTRY, ACTIVATE, DEACTIVATE, SWAP_IN, and SWAP_OUT. These instructions correspond one to one with those of the Distributed Memory Manager Module. The system wide data bases utilized by all Memory Manager Processes are the Global Active Segment Table (G_AST), the Alias Table, the Disk Bit Map, and the Global Memory Bit Map. The processor local databases used by each Memory Manager Process are the Local Active Segment Table (L_AST), and the Local Memory Bit Map. Gary and Moore [5] provide a detailed description of the Memory Manager, its extended instruction set, and its databases.

A summary of the extended instruction set created by the components of the Security Kernel is provided by Figure 10. One might question the prudence of omitting PHYS_PREEMPT_HANDLER and VIRT_PREEMPT_HANDLER (viz., the handler routines for physical and virtual interrupts) from the extended instruction set as both of these interrupts may be raised (viz., initiated) from within the Kernel. A decision was made to not classify these handlers as "extended instructions" since they are only executed as the result of a physical or virtual interrupt and as such cannot be directly invoked (viz., "called") by any module in the system.

A summary of the databases utilized by Kernel modules is presented in Figure 11.

MODULE	INSTRUCTION SET					
Segment Manager	Create_Segment*	Delete_Segment*				
	Make_Known*	Terminate*				
	SM_Swap_In*	SM_Swap_Out*				
Event Manager	Read*	Ticket*				
	Advance*	Await*				
Non-Discretionary Security	Class_EQ	Class_GE				
Traffic Controller	TC_Advance	TC_A wait				
	Process_Class					
Inner Traffic	Signal	Wait				
Controller	Swap_VDBR	Idle				
	Set_Preempt	Test_Preempt				
	Running_VP					
Distributed	MM_Create_Entry	MM_Delete_Entry				
Memory Manager	MM_Activate	MM_Deactivate				
	MM_Swap_In	MM_Swap_Out				
Non-Distributed	Create_Entry	Delete_Entry				
Memory Manager	Activate	Deactivate				
	Swap_In	Swap_Out				

* Denotes user visible instructions

Figure 10: Extended Instruction Set

MODULE

DATABASE

Gate Keeper

Parameter Table

Segment Manager

Known_Segment_Table (KST)

Traffic Controller

Active_Process_Table (APT)

Inner Traffic

Virtual_Processor_Table (VPT)

Controller

Memory_Management_Unit Image

(MMŪ)

Memory Manager

Global_Active_Segment_Table (G_AST)

Local_Active_Segment_Table (L_AST)

Disk_Bit_Map

Global_Memory_Bit_Map

Local_Memory_Bit_Map

Figure 11: Kernel Databases

Chapter VIII SYSTEM HARDWARE

Level 0 of the SASS consists of the system hardware. This hardware includes: 1) the CPU, 2) the local memory, 3) the secondary storage (viz. hard the global memory, 4) disk), and 5) the I/O ports connecting the Host computer systems to the SASS. Since the SASS design allows for a multiprocessor environment, there may exist multiple CPU's and local memories. The target machine selected for the initial implementation of the system is the Zilog Z8001 microprocessor [22]. The Z8001 is a general purpose 16-bit, register oriented machine that has sixteen 16-bit general purpose registers. It can directly address 8M bytes of memory, extensible to 48M bytes. The Z8001 architecture supports memory segmentation and two-domain operations. The memory segmentation capability is provided externally by the The Z8010 MMU Zilog Z8010 Memory Management Unit (MMU). [23] provides management of the Z8001 addressable memory, dynamic segment relocation, and memory protection. Memory segments are variable in size from 256 bytes to 64K bytes and are identified by a set of 64 Segment Descriptor Registers, which supply the information needed to map logical memory addresses to physcal memory addresses. Each of the 64

Descriptor Registers contains a 16-bit base address field, an 8-bit limit field, and an 8-bit attribute field. tunately, the 28001 hardware was not available for use during system development. Therefore, all work to date has been completed through utilization of the Z8002 non-segmented version of the Z8000 microprocessor family [22]. The actual hardware used in this implementation is the Advanced Micro Computers Am 96/4116 MonoBoard Computer [1] containing the Amz8002 sixteen bit non-segmented microprocessor. computer provides 32K bytes of on-board RAM, 8k bytes of PROM/ROM space, two RS232 serial I/O ports, 24 parallel I/O lines, and a standard INTEL Multibus interface. The general structure of the design has been preserved by simulation of the segmentation hardware in software. This software MMU Image (see Figure 12) is created as a database within the Inner Traffic Controller.

The MMU Image is a processor-local database indexed by DBR_No. Each DBR_No represents one record within the MMU Image. Each record is an exact software copy of the Segment Descriptor Register set in the hardware MMU. Each element of this software MMU Image is in the same form utilized by the special I/O instructions to load the hardware MMU. Each DBR record is indexed by segment number (Segment_No). Each Segment_No entry is composed of three fields: Base_Addr, Limit, and Attributes. Base_Addr is a 16-bit field which contains the base address of the segment in physical memory.

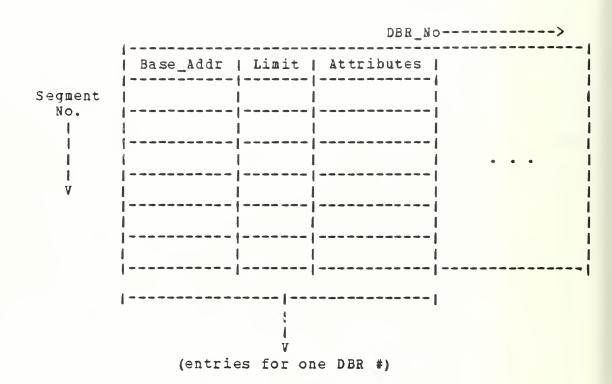


Figure 12: Memory Management Unit (MMU) Image

Limit is an 8-bit field that specifies the number of contiguous blocks of memory occupied by the segment. Attributes is an 8-bit field representing the eight flags which specify the segment's attributes (e.g., "read", "execute", "write", etc.).

Chapter IX

SUMMARY

An extended overview of the current SASS design has been presented. The four major levels of abstraction comprising the SASS system have been identified, and the major components of each level have been discussed. The extended instruction set provided by the SASS Kernel was also defined. The actual details of this implementation are described by Strickler [19].

PART C

THE DESIGN AND IMPLEMENTATION OF THE MEMORY MANAGER FOR A SECURE ARCHIVAL STORAGE SYSTEM

This section contains updated excerpts from a Naval Postgraduaduate School MS Thesis by E. E. Moore and A. V. Gary [5]. The origins of these excerpts are:

INTRODUCTION from Chapter I
MEMORY MANAGER PROCESS DETAILED DESIGN from Chapter III
STATUS OF RESEARCH from Chapter IV

Minor changes have been made for integration into this report.

Chapter X

INTRODUCTION

This thesis addresses the design and partial implementation of a memory manager for a member of the family of secure, distributed, multi-microprocessor operating systems designed by Richardson and O'Connell [7]. The memory manager is responsible for the secure management of the main memory and secondary storage. The memory manager design was approached and conducted with distributed processing, multi-processing, configuration independence, ease of change, and internal computer security as primary goals. The problems faced in the design were:

- 1) Developing a process which would securely manage files in a multi-processor environment.
- 2) Ensuring that if secondary storage was inadvertantly damaged, it could usually be recreated.
- 3) Minimizing secondary storage accesses.
- 4) Proper parameter passing during interprocess communication.
- 5) Developing a process with a loop-free structure which is configuration independent.

6) Designing databases which optimize the memory management functions.

The proper design and implementation of a memory management process is vital because it serves as the interface between the physical storage of files in a storage system and the logical hierarchical file structure as viewed by the user (viz., the file system supervisor design by Parks [9]. If the memory manager process does not function properly, the security of that system cannot be quaranteed.

The secure family of operating systems designed by Richardson and O'Connell is composed of two primary modules, the supervisor and the security kernel. A subset of that system was utilized in the design of the Secure Archival Storage System (SASS). The design of the SASS supervisor was addressed by Parks [9], while the security kernel was addressed concurrently by Coleman [2]. The SASS security kernel design is composed of two parts, the distributed kernel and the non-distributed kernel. The design of the distributed kernel was conducted by Coleman [2], and processor management was implemented by Reitz [12]. This thesis presents the design and implementation of the non-distributed kernel. In the SASS design, the non-distributed kernel consists solely of the memory manager.

The design of the memory manager and its data bases was completed. The initial code was written in PLZ/SYS, but could not be compiled due to the lack of a PLZ/SYS compiler.

A thread of the high level code was selected, hand compiled into PLZ/ASM, and run on the Z8000 developmental module.

Chapter XI

MEMORY MANAGER PROCESS DETAILED DESIGN

A. INTRODUCTION

The memory manager is responsible for the management of both main memory (local and global) and secondary storage. It is a non-distributed portion of the kernel with one memory manager process existing per physical processor. The memory manager is tasked (via signal and wait) to perform memory management functions on behalf of other processes in the system. The major tasks of the memory manager are: 1) the allocation and deallocation of secondary storage, 2) the allocation and deallocation of global and local memory, 3) segment transfer from local to global memory (and vice versa), and 4) segment transfer from secondary storage to main memory (and vice versa). There are ten service calls (via signal) which task the memory manager Process to perform these functions. The ten service calls are:

CREATE_ENTRY
DELETE_ENTRY
ACTIVATE
DEACTIVATE
SWAP_IN
SWAP_OUT
DEACTIVATE_ALL*
MOVE_TO_GLOBAL*
MOVE_TO_LOCAL*
UPDATE*

Upon completion of the service request, the memory manager returns The results of the operation to the waiting process (via signal). It then blocks itself until it is tasked to perform another service. The hardware configuration managed by the memory manager process is depicted in Figure 13. The shared data bases used by all memory manager processes are the Global Active Segment Table (G_AST), the Alias Table, the Disk Bit Map, and the Global Memory Bit Map. The processor local data bases used by each process are the Local Active Segment Table (L_AST), the Memory Management Unit Images and the Local Memory Bit Map.

^{*} In the current state these service calls are not implemented therefore, there are currently six service calls.

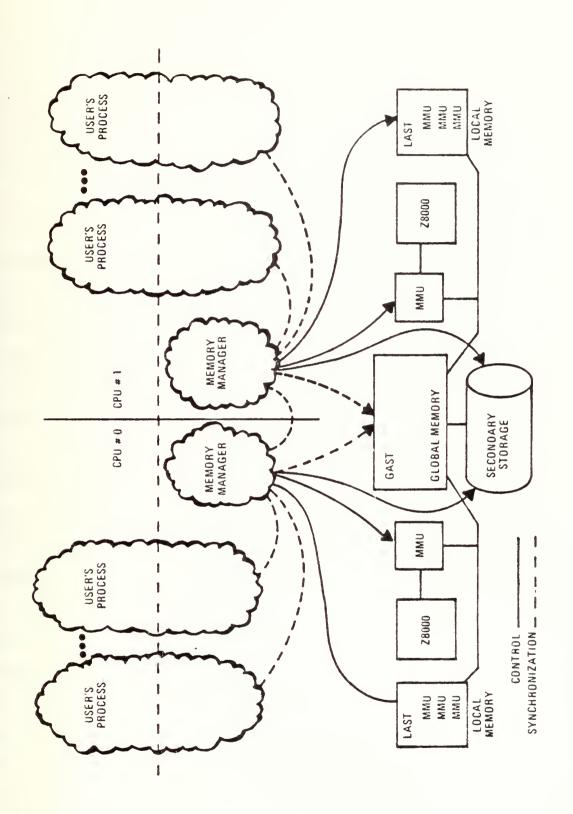


Figure 13: SASS H/W System Overview

B. DESIGN PARAMETERS AND DECISIONS

Several factors were identified during the design of the memory manager process that refined the initial kernel design of Coleman [2]. The two areas that were modified, were the management of the MMU images and the management of core memory. Both of these functions were managed outside of the memory manager in the initial design. The inclusion of these functions in the memory manager process significantly improved the logical structure of the overall system design. Additional design parameters were established to facilitate the initial implementation. These design parameters need to be addressed before the detailed design of the memory manager process is presented.

It was decided to make the block/page size of both main memory and secondary storage equal in size. This was to simplify the mapping algorithm from secondary storage to main memory (and vice versa). In the initial design the block/page size was set to 512 bytes.

The size of the page table for a segment was set at one page (non-paged page table). This was to simplify implementation, and had a direct bearing on the maximum segment size supported in the memory manager. For example, a page size of 256 bytes will address a maximum segment size of 32,768 bytes, while a page size of 512 bytes will address a segment size of 131,072 bytes.

The size of the alias table was set to one page (non-paged alias table). The number of entries that the alias table will support is limited by the size of the page table (viz., a page size of 512 bytes will support up to 42 entries in the Alias Table).

In the original design, the main memory allocation was external to the memory manager. This was due to the partitioned memory management scheme outlined by Parks [9] and Coleman [2]. In the current design, all address assignment and segment transfer are managed by the memory manager. This design choice enhanced the generality of the design, and provided support for any memory management scheme (either in the memory manager or at a higher level of abstraction). However, the current design still has a maximum core constraint for each process.

Dynamic memory management is not implemented in this design. Each process is allocated a fixed size of physical core. However, it is not a linear allocation of physical memory. The design supports the maximum sharing of segments in local and global memory. All segments that are not shared, or shared and do not viciate the readers/writers problem will reside in local memory to eliminate the global bus contention. The need to compact the memory (because of fragmentation) should be minimal in this design due to the maximum sharing of segments. If contiguous memory is not available, the memory manager will compact main memory. After compaction, the memory can be allocated.

The design decision to represent memory as one contiguous block (not partitioned) was made to support a dynamic memory management scheme. Without dynamic memory management, the process total physical memory can not exceed the systems main memory. The supervisor knows the size of the segments and the size of the process virtual core, therefore it can manage the swap in and swap out to ensure that the process virtual core has not been exceeded.

In the original design, the user's process inner-traffic controller maintained the software images of the memory management unit. This design required the memory manager to return the appropriate memory management data (viz., segment location) to the kernel of the user's process. In the current design, the software images of the MMU are maintained by the memory manager. A descriptor base pointer is provided for the inner-traffic controller to multiplex the process address spaces. The MMU image data base does not need to be locked (to prevent race conditions) due to the fact that process interrupts are masked in the kernel. Thus, if the memory manager (a kernel process) is running then no other process can access the MMU image.

The system initialization process has not been addressed to date. However, this design has made some assumptions about the initial state of the system. Since the memory manager handles the transfer of segments from secondary storage to main memory, it is likely to be one of the first

processes created. The memory manager's core image will consist of its pure code and data sections. The minimal initialization of the memory manager's data bases are entries for the system root and the supervisor's segments in the G_AST and L_AST(s), and the initialization of the MMU images with the kernel segments. The current design does not call for an entry in the G_AST or L_AST for the kernel segments. However, when system generation is designed this will have to be readdressed.

The original [2] memory manager databases have been refined by this thesis to facilitate the memory management functions. The major refinements of the global and local active segment tables are outlined in the following section.

C. DATA BASES

1. Global Active Segment Table

The Global Active Segment Table (see Figure 14) is a system wide, shared data base used by memory manager processes to manage all active segments. A lock/unlock mechanism is utilized to prevent any race conditions from occurring. The signalling process locks the G_AST before it signals the memory manager. This is done to prevent a deadly embrace from occurring between memory manager processes, and also to simplify synchronization between memory managers. The entire G_AST is locked in this design to simplify the implementation (vice locking each individual entry).

Ind	ex_#					
1	Unique	 Global Addr	* Processors L_ASTE_#	Flag		/ G_ASTE_#/ Parent/
1			#0			<u> </u>
A		 		1 1 1		
					! 	/

* Field indicates a two processor environment

/ # Active /In Memory /	Activ	e Siz		Tablequenc	
/ / / /		1		8 8 4	
, ,		<u> </u>	i i		

Figure 14: Global Active Segment Table

The G_AST size is fixed at compile time. The size of the G_AST is the product of the G_AST record size, the maximum number of processes and the number of authorized known segments per process. Although the G_AST is of fixed size, it is plausible to dynamically manage the entries as proposed by Richardson and O'Connell [7]. The current memory manager design could be extended to include this dynamic management.

The Unique_Id field is a unique segment identification number in the G_AST. This field is four bytes wide and will provide over four billion identification numbers. A design choice was made not to manage the reallocation of the unique_id's. Thus when a segment is deleted from the system, the unique_id is not reused.

The Global_Address field is used to indicate if a segment resides in global or local memory. If not null, it contains the global memory base address of a segment. A null entry indicates that the segment might be in local memory(s).

The Processors_L_ASTE_# field is used as a connected processors list. The field is an array structure, indexed by Processor_Id. It identifies which L_AST the segment is active in, and provides the index into each of these tables. The design choice of maintaining an entry in the L_AST for all locally active segments implies that if all entries in the Processors_L_ASTE_# field are null, the segment is not

active and can be removed from the G_AST (viz., no processors are connected).

The Flag_Bits field consists of the written bit, and the writable bit. The written bit is set when a segment is swapped out of memory, and the MMU image indicates that it has been written into. The writable bit is set during segment loading to indicate that some process has write access to that segment.

If an active segment is a leaf, the G_ASTE_#_Parent field provides a back pointer to the G_AST index of its parent. This back pointer to the parent is important during the creation of a segment. If a request is received to create a segment which has a leaf segment as its parent, then an alias table has to be created for that parent. Also, the alias table of the parent's parent needs to be updated to reflect the existence of the newly created alias table (see Figure 15). The indirect pointer shown is the back pointer to the parent via the G_AST.

The No_Active_In_Memory field is a count of the number of processes that have the segment in global memory. It is used during swap out to determine if the segment can be removed from global memory.

The No_Active_Dependents field is a count of the number of active leaf segments that are dependent on this entry (viz., require that this segment remain in the G_AST). Each time a process activates or deactivates a dependent segment this field is incremented or decremented.

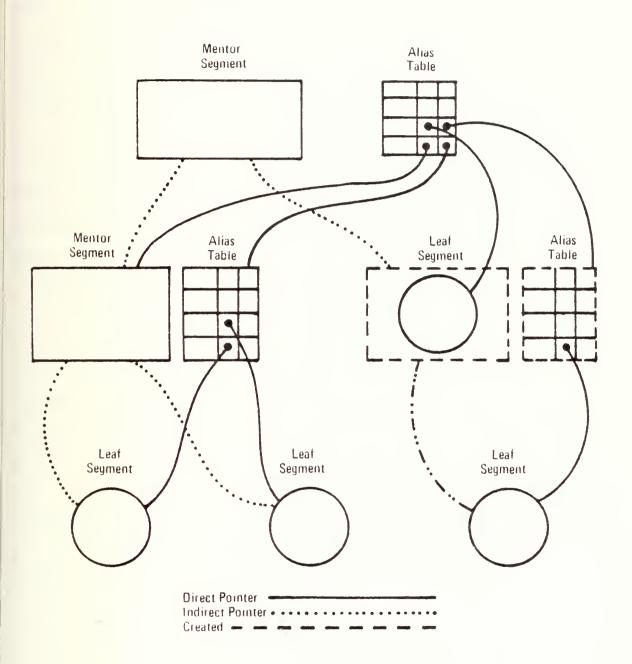


Figure 15: Alias Table Creation

The Size field is the size of the segment in bytes. The Page_Table_location field is the disk location of the page table for a segment, and the Alias_Table_Location field is the disk location of the alias table for the segment. The Alias_Table field can be null to indicate that no alias table exists for the segment.

The last three fields are used in the management of eventcounts and sequencers [12]. The Sequencer field is used to issue a service number for a segment. The Instance_1 field and Instance_2 field are eventcounts (i.e., are used to indicate the next number of occurances of some event).

2. Local Active Segment Table

The Local Active Segment Table (see Figure 16) is a processor local data base. The L_AST contains the characteristics (viz., segment number, access) of each locally active segment. An entry exists for each segment that is active in a process "loaded" on this CPU and in local memory. The first field of the L_AST contains the memory address of the segment. If the segment is not in memory, this field is used to indicate whether the L_AST entry is available or active. The Segment_No/Access field is a combination of segment number and authorized access. It is an array of records data structure that is indexed by DBR_#. The first record element (viz., most significant bit) is used to indicate the access (read or read/write) permitted to that segment. The

second record element (viz., the next seven bits) is used to indicate the segment number. A null segment number indicates that the process does not have the segment active.

Index_#

	Memory	Segment_#/Access_Auth						
	Addr	DBR_O	DBR_1	DBR_2	DBR_3	DBR_4	DBR_5	
1								
A								

Figure 16: Local Active Segment Table

3. Alias Table

The alias table (see Figure 17) is a memory manager data base which is associated with each non leaf segment in the kernel. An aliasing scheme is used to prevent passing systemwide information (unique_id.) out of the kernel. Segments can only be created through a mentor segment and entry

number into the mentor's alias table. When a segment is created, an entry must be made in its mentor segment's alias table. Thus the mentor segment must be known before that segment can be created.

Enti	ry_#				
	Unique_ID	Size	Class	Page Table Location	Alias Table Location
1 1		 	 	1 !	1
▼ i		1		1	
1		1 1 1	 	1 1 1	
i		i 	i 	i 	i i

Figure 17: Alias Table

The alias table consists of a header and an array structure of entries. The header has two "pointers" (viz., disk addresses), one that links the alias table to its associated segment and one that links the alias table to the mentor segment's alias table. The header is provided to support the re-construction of the file system after a system crash due

to device I/O errors. It is not used at all during normal operations. Each entry in the array structure consists of five fields for identifying the created segments. The Unique_Id field contains the unique identification number for the segment. The Size field is used to record the size of the segment. The Class field contains the appropriate security access class of the segment. The Page_Table_Location field has the disk address of the page table. A null entry indicates a zero-length segment. The Alias_Table_Location field has the disk address of the alias table for the segment. A null entry indicates that the segment is a leaf segment.

4. Memory Management Unit Image

The Memory Management Unit Image (MMU_Image) is a processor local data base. It is an array structure that is indexed by the DBR_#. Each MMU_Image (see Figure 18) includes a software representation of the segment descriptor registers (SDR) for the hardware MMU [23]. This is in exactly the format used by the special I/O instructions for loading/unloading the MMU hardware. The SDR contains the Base_Address, Limit and Attribute fields for each loaded segment in the process address space. The Base_Address field contains the base address of the segments in memory (local or global). The Limit field is the number of blocks of contiguous storage for each segment (zero indicates one

block). The Attribute field contains eight flags. Five flags are used for protecting the segment against certain types of access, two encode the type of accesses made to the segment (read/write), and one indicates the special structure of the segment [23]. Five of the eight flags in the attribute field are used by the memory manager. The "system only" and "execute only" flags are used to protect the code of the kernel from malicious or unintentional modifications. The "read only" flag is used to control the read or write access to a segment. The "change" flag is used to indicate that the segment has been written into, and the "CPU-inhibit" flag is used to indicate that the segment is not in memory.

The last two fields of the MMU_Image are the Block_Used field and the Maximum_Available_Blocks field. These two fields are used in the mangement of each process virtual core and are not associated with the hardware MMU.

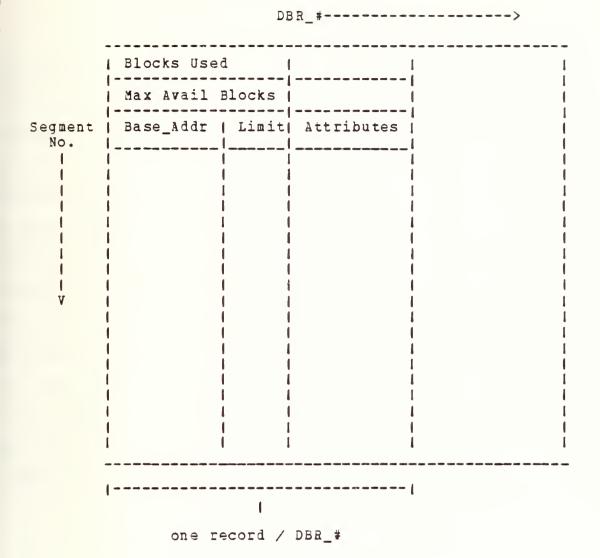


Figure 18: Memory Management Unit Image

5. Memory Allocation/Deallocation Bit Maps

All of the memory allocation/deallocation bit maps (see Figure 19) are basically the same structure. Secondary storage, global memory and local memory are managed by memory The Disk_Bit_Map is a global resource that is protected from race conditions via the locking convention for the G_AST. Each bit in the bit map is associated with a block of secondary storage. A zero indicates a free block of storage while a one indicates an allocated block of storage. The Global_Memory_Bit_Map is used to manage global memory. It is a shared resource that is protected from race conditions by the locking of the G_AST. The Local_Memory_Bit_Map is the same structure as the Global_Memory_Bit_Map and is used to manage local memory. The Local_Memory_Bit_Map is not locked since it is not a shared resource between memory managers.

Memory Bit Map

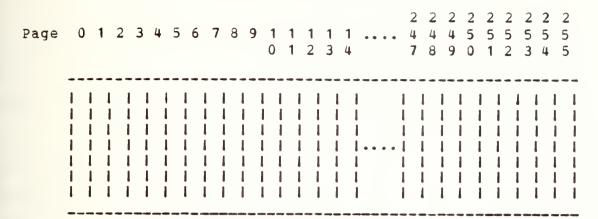


Figure 19: Memory Allocation/Deallocation Map

D. BASIC PUNCTIONS

The detailed source code for the basic functions and main line of the memory manager is presented in Appendix J.

In the discussion of the memory manager design, a pseudo-code similar to PLZ/SYS is utilized. The rationale for using this pseudo-code was to provide a summary of the memory manager source code, and to facilitate the presentation of this design.

It is assumed that the memory manager is initialized into the ready state at system generation (as previously mentioned). When the memory manager is initially placed into the running state, it will block itself (via a call to

the kernel primitive Wait). Wait will return a message from a signalling process. This message is interpreted by the memory manager to determine the requested function and its required arguments. The function code is used to enter a case statement, which directs the request to the appropriate memory manager procedure.

When the requested action is completed, the memory manager returns a success code (and any additional required data) to the signalling process via a call to the kernel primitive Signal. This call will awaken the process which requested the action to be taken, and place the returned message into that process message queue. When that action is completed, the memory manager will return to the top of the loop structure and block itself to wait for the the next request. The main line pseudo-code of the memory manager process is displayed in Figure 20.

```
ENTRY
    INITIALIZE PROCESSOR LOCAL VARIABLES
    DO
           CHECK_IF_MSG_QUEUE_EMPTY
        VP_ID, MSG := WAIT
        FUNCTION, ARGUMENTS := VALIDATE MSG (MSG)
        IF
            FUNCTION
            CASE CREATE_ENTRY THEN
                SUCCESS CODE := CREATE ENTRY (ARGUMENTS)
            CASE DELETE ENTRY THEN
                SUCCESS CODE := DELETE_ENTRY (ARGUMENTS)
                  ACTIVATE THEN
            CASE
                SUCCESS CODE := ACTIVATE (ARGUMENTS)
            CASE DEACTIVATE THEN
                SUCCESS_CODE := DEACTIVATE (ARGUMENTS)
            CASE SWAP_IN THEN
                SUCCESS_CODE := SWAP_IN (ARGUMENTS)
                  SWAP_OUT THEN
                SUCCESS CODE := SWAP OUT (ARGUMENTS)
            CASE DEACTIVATE_ALL THEN
                SUCCESS_CODE := DEACTIVATE_ALL (ARGUMENTS)
                 MOVE_TO_GLOBAL THEN
                SUCCESS CODE := MOVE TO GLOBAL (ARGUMENTS)
            CASE MOVE_TO_LOCAL THEN
SUCCESS_CODE := MOVE_TO_LOCAL (ARGUMENTS)
            CASE UPDATE
                          THEN
                SUCCESS CODE := UPDATE (ARGUMENTS)
        SIGNAL (VP_ID, SUCCESS_CODE, ARGUMENTS)
    OD
END MEMORY_MANAGER_PLZ/SYS MODULE
```

Figure 20: Memory Manager Mainline Code

1. Create an Alias Table Entry

Create_Entry is invoked when a user desires to create a segment. A segment is created by allocating secondary storage, and by making an entry (unique_id, secondary storage location, size, classification) into it's mentor segment's alias table. This implies that the mentor segment must have an alias table associated with it, and that the mentor segment must be active in order to obtain the secondary storage location of the alias table.

The mentor segment can be in one of two states. It may have children (viz., have an alias table), or it may be a leaf segment (viz., not have an alias table). If the mentor segment has children, it has an alias table and this alias table can be read into core, secondary storage can be allocated, and the data can be entered into the alias table. If the mentor segment is a leaf, an alias table must be created for that segment before it (the alias table) can be read into core and data entered into it (see Figure 15).

The pseudo-code for CREATE_ENTRY PROCEDURE is presented in Figure 21. The arguments passed to Create_Entry are the index into the G_AST for the mentor segment, the entry number into its alias table, the size of the segment to be created, and the security access class of that segment. The return parameter is a success code, which would be "seg_created" for a successful segment creation.

```
CREATE_ENTRY PROCEDURE (PAR_INDEX WORD, ENTRY_# WORD,
                          SIZE
                                 WORD, CLASS
   RETURNS (SUCCESS CODE
                           BYTE)
   LOCAL BLKS WORD, PAGE_TABLE_LOC WORD
   ENTRY
   IF
       ALIAS_TABLE_DOES_NOT_EXIST
       SUCCESS CODE := CREATE ALIAS TABLE
           SUCCESS CODE <> VALID THEN RETURN
       FI
   FI
   BLKS := CALCULATE_NO_BLKS_REQ (SIZE)
  SUCCESS_CODE := READ_ALIAS_TABLE (
                    G_AST[ PAR_INDEX ]. ALIAS_TABLE_LOC)
       SUCCESS_CODE
                    <> VALID THEN RETURN
   IF
   FI
   SUCCESS_CODE := CHECK_DUP_ENTRY ! in alias table !
   IF SUCCESS CODE <> VALID THEN RETURN
   SUCCESS_CODE, PAGE_TABLE_LOC := ALLOC_SEC_STORAGE (BLKS)
   IF SUCCESS CODE <> VALID
                                 THEN
                                        RETURN
   FI
  UPDATE_ALIAS_TABLE(ENTRY_#, SIZE, CLASS, PAGE_TABLE_LOC)
SUCCESS_CODE := WRITE_ALIAS_TABLE (
                          G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
   IF SUCCESS_CODE <>
                      VALID
                               THEN
                                       RETURN
   ELSE SUCCESS_CODE := SEG_CREATED
   FI
```

END CREATE_ENTRY

Figure 21: Create Entry Pseudo-code

when invoked, Create_Entry will determine which state the mentor segment is in (viz., if it has an alias table). If an alias table does not exist for the mentor segment, one is created and the alias table of the mentor segment's parent is updated. The alias table is read into core and a duplicate entry check is made. If no duplicate entry exists, the segment size is converted from bytes to blocks, and the secondary storage is allocated for non-zero sized segments. The appropriate data is entered into the alias table and the alias table is then written back to secondary storage.

2. Delete an Alias Table Entry

Delete_Entry is invoked when a user desires to delete a segment. A segment is deleted by deallocating secondary storage, and by removing the appropriate entry from the alias table of its mentor segment (the reverse logic of Create_Entry). This implies that the mentor segment must be active at the time of deletion. There are three conditions that can be encountered during the deletion of a segment: the segment to be deleted may be an inactive leaf segment, an active leaf segment, or a mentor segment.

If the segment to be deleted is an inactive leaf segment (viz., has been swapped out of core, and does not have an entry in the G_AST), the secondary storage can be deallocated and the entry deleted from the mentor segment's alias table. If the segment is an active leaf segment, the segment

must first be swapped out of core and deactivated before it can be deleted. This entails signalling the memory manager of each processor, in which the segment is active, to swap out and deactivate the segment.

If the segment to be deleted is a mentor segment, an alias table exists for that segment. If the alias table is empty, the secondary storage for the alias table and the segment can be deallocated, and the entry for the deleted segment can be removed from its mentor's alias table. If the alias table contains any entries, the segment cannot be deleted because these entries would be lost. If this condition is encountered a success code of "leaf_segment_exists" is returned to the process which requested to delete the entry. Due to a confinement problem in "upgraded" segments, this success_code cannot always be passed outside of the kernel. This implies that the segment manager must strictly prohibit deletion of a segment with an access class not equal to that of the process.

The pseudo-code for DELETE_ENTRY_PROCEDURE is presented in Figure 22. The parameters that are passed to this procedure are the parent's index into the G_AST and the entry number into the parent's alias table of the segment to be deleted. The alias_table_loc field is checked to determine the state of the mentor segment (either a leaf or a node), and the appropriate action is then taken. A success code is returned to indicate the results of this procedure.

```
DELETE_ENTRY PROCEDURE ( PAR_INDEX WORD, ENTRY_# WORD )
   RETURNS (SUCCESS_CODE
                          BYTE)
   LOCAL
         PAR_INDEX WORD
   ENTRY
 ! Check if the passed mentor segment has an alias table. !
   IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC <> NULL
       SUCCESS_CODE := READ_ALIAS_TABLE (
                       G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
   ELSE
       SUCCESS_CODE := NO_CHILD_TO_DELETE
   ΡI
   IF
       SUCCESS CODE <> VALID THEN
                                         RETURN
   FI
 ! Determine if segment has children in alias table !
       ALIAS_TABLE_NOT_EMPTY
       SUCCESS_CODE := LEAF_SEGMENT_EXISTS
       RETURN ! Deletion will delete children !
   ELSE
! Search G_AST with UNIQUE_ID to verify segment inactive !
           ACTIVE_IN_G_AST THEN
           ! Check if active in AST !
               ACTIVE IN_L_AST THEN
           ΙF
               DEACTIVATE_ALL (G_AST_INDEX, L_AST_INDEX)
           PI
! Check G_AST to verify segment inactive in other L AST's !
               ACTIVE_IN_OTHER_L_AST THEN
           IF
               SIGNAL_TO_DEACTIVATE_ALL (G_AST_INDEX)
           FI
       FI
       FREE_SEC_STORAGE_OF_SEG_&_ALIAS_IF_EXISTS
       DELETE_ALIAS_TABLE_ENTRY
   FI
   DELETE_ALIAS_TABLE_ENTRY
   SUCCESS_CODE := WRITE_ALIAS_TABLE (
                     G_AST[ PAR_INDEX ].ALIAS TABLE LOC)
       SUCCESS CODE = VALID
   IF
                            THEN
        SUCCESS_CODE := SEG_DELETED
   FI
END DELETE_ENTRY
```

Figure 22: Delete Entry Pseudo-code

3. Activate a Segment

Activate is invoked when a user desires to make a segment known by adding a segment to his address space. A segment is activated by making an entry into the L_AST for that processor, and the G_AST. The activated segment could be in one of three states; it could have previously been activated by another process and have a current entry in both the G_AST and L_AST, it could have previously been activated by another process on a different processor and have an entry in the G_AST but not the L_AST, or it could be inactive and have an entry in neither the G_AST nor the L_AST.

If the segment to be activated already has entries in both the L_AST and G_AST, these entries need only be updated to indicate that another process has activated the segment. The segment number is entered into the Segment_No/Access_Auth field of the L_AST, and if the segment is a leaf, its mentor's No_Active_Dependents field in the G_AST is incremented. In this design, the G_AST is always searched to determine if the segment has been previously activated by another process.

If the segment to be activated has an entry in the G_AST but not the L_AST, an entry must be made in the L_AST and the G_AST must be updated. The L_AST is searched to determine an available index. The segment number is entered into the L_AST, and the index number is entered into the G_AST

Processors_L_ASTE_# field. If the segment to be activated is a leaf segment, its mentor's No_Active_Dependents field in the G_AST is incremented.

If the activated segment does not have an entry in either the G_AST or L_AST, an entry must be made in both. The G_AST is searched to find an available index, and the entry is made. The L_AST is then searched to find an available index, and the entry is made. The L_AST index is then entered into the G_AST Processors_L_ASTE_# field. If the activated segment is a leaf, the No_Active_Dependents field of its mentor's G_AST entry is incremented.

The pseudo-code for ACTIVATE PROCEDURE is presented in Figure 23. The parameters that are passed are the DBR_* of the signalling process, the mentor segment's index into the G_AST, the alias table entry number, and the segment number of the activated segment. The mentor segment is always checked to determine if it has an associated alias table. If it does not, the success code of "alias_does_not_exist" is returned. If the alias table does exist, it is read into core and the entry number is used as an index to obtain the activated segment's unique_id. The G_AST is then searched to determine if the segment has already been activated. If the unique_id is found, the G_AST is updated and the L_AST is either updated or an entry is made (depending on whether an entry existed or not). If the unique_id of the segment was not found during the search of the G_AST, an entry must

be made in both the G_AST and L_AST. Activate returns the activated segment's classification, size, and handle to the signalling process.

```
PROCEDURE (DBR_# BYTE, PAR_INDEX WORD,
ACTIVATE
                      ENTRY # WORD, SEGMENT_NO BYTE)
    RETURNS (SUCCESS_CODE BYTE, RET_G_AST_HANDLE HANDLE,
                       CLASS BYTE, SIZE WORD)
                     WORD, L INDEX
    LOCAL
            G INDEX
                                    WORD
    ENTRY
  ! Verify that passed segment is a mentor segment !
    IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC <> 0 THEN
        SUCCESS_CODE := READ_ALIAS_TABLE (
                          G AST[PAR_INDEX].ALIAS_TABLE_LOC)
    ELSE
        SUCCESS CODE := ALIAS DOES NOT EXIST
    FI
    IF
        SUCCESS_CODE <> VALID
                               THEN
                                       RETURN
    FI
  ! Check G_AST to determine if active !
    SUCCESS_CODE, INDEX := SEARCH_G_AST (UNIQUE_ID)
    IF SUCCESS CODE = FOUND THEN
            SEGMENT_IN_L_AST THEN
        IF
            UPDATE L AST (SEGMENT NO)
        ELSE
            MAKE_L_AST_ENTRY (DBR_#, SEGMENT_NO)
            UPDATE_G_AST (L_INDEX)
            IF G_AST[INDEX].ALIAS_TABLE_LOC = NULL THEN
                G AST[PAR INDEX]. NO DEPENDENTS ACTIVE += 1
            FI
        PI
    ELSE
        MAKE_G_AST_ENTRY (ENTRY_#)
        MAKE_L_AST_ENTRY (PAR_INDEX, ENTRY #)
    SUCCESS_CODE := SEG ACTIVATED
END ACTIVATE
```

Figure 23: Activate Pseudo-code

4. Deactivate a Segment

Deactivate is invoked when a user desires to remove a segment from his address space. To deactivate a segment, the memory manager either removes or updates an entry in both the L_AST and G_AST. Deactivate uses the reverse logic of activate. Once a segment is deactivated, it can only be reactivated via its mentor's alias table as discussed in activate. If a process requests to deactivate a segment which has not been swapped out of the process' virtual core, the memory manager swaps the segment out and updates the MMU image before the segment is deactivated. The segment to be deactivated could be in one of three states; more than one process could concurrently hold the segment active in the L_AST, the segment could be held active by one process in the L_AST and more than one in the G_AST, the segment could be held active by only one process in both the L_AST and the G AST.

Deactivation of leaf segments and mentor segments are handled differently. If the segment is a mentor segment and has active dependents, it cannot be removed from the G_AST (even though no process currently has that segment active). This is based on the design decision which requires that the mentor of all active leaf segments remain in the G_AST to allow access to its alias table. The mentor's alias table must be accessible when an alias table is created for a de-

pendent leaf segment. If a leaf segment is deactivated, the No_Active_Dependents field of its mentor's G_AST entry is decremented. A mentor segment can only be removed from the G_AST if no process holds it active, and it has no active dependents.

If more than one process concurrently hold a segment active in the L_AST, and one of them signals to deactivate that segment, the entry in the L_AST is updated. This is accomplished by nulling out the Segment_No/Access_Auth field of the L_AST for the appropriate process. If required, the No_Active_Dependents field of its mentor segment's G_AST entry is decremented.

only one process holds the segment active in the L_AST, and that Process signals to deactivate the segment, the L_AST entry for that segment is removed. The Processors_L_ASTE_# is updated and checked to determine if there are other connected processors. If there are no other connected processors and the segment has no active dependents, the segment is removed from the G_AST. If there are other connected processors, the G_AST is updated. If the deactivated seqment is a leaf, the mentor segment's No_Active_Dependents field in the G_AST is decremented.

The pseudo-code for DEACTIVATE PROCEDURE is presented in Figure 24. The parameters that are passed to the memory manager are the DBR_# of the signalling process, and the index into the G_AST for the segment to be deactivated. The

procedure first updates the L_AST, and then removes the entry if no local process holds the segment active. The G_AST is then updated, and its mentor segment is checked (if the deactivated segment was a leaf), to determine if it can be removed. If no processes currently hold the segment active, and it has no active dependents, the segment is removed from the G_AST.

```
DEACTIVATE PROCEDURE (DBR_# BYTE, PAR_INDEX WORD)
    RETURNS (SUCCESS_CODE BYTE)
    LOCAL
           INDEX
                   WORD
    ENTRY
  ! Check if segment is in core !
    IF G_AST[INDEX]. NO_ACTIVE_IN_MEMORY <> 0 THEN
       ! Check MMU image to determine if in local memory !
            IN_LOCAL_MEMORY THEN
            SUCCESS CODE := OUT (DBR_#, INDEX)
        FI
    PI
  ! Remove process segment_no entry in L_AST !
    L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH[DBR_#] = 0
    CHECK_IF_ACTIVE_IN_L_AST (L_AST_INDEX)
        NOT_ACTIVE_IN_L_AST THEN
        L AST[L INDEX]. MEMORY ADDR := AVAILABLE
  ! Check if deleted segment was a leaf!
    IF G_AST[INDEX].G ASTE # PAR <> 0 THEN
        G_AST[PAR INDEX].NO_DEPENDENTS_ACTIVE -= 1
  ! Determine if parent can be removed!
       CHECK_FOR_REMOVAL (PAR_INDEX)
    FI
  ! Determine if deactivated segment can be removed!
    CHECK_FOR_REMOVAL (INDEX)
    SUCCESS CODE := SEG DEACTIVATED
END DEACTIVATE
```

Figure 24: Deactivate Pseudo-code

5. Swap a Segment In

SWAP_IN is invoked when a user desires to swap a segment into main memory (global or local) from secondary storage. A segment is swapped into main memory by obtaining the secondary storage location of its page table from the G_AST, allocating the required amount of main memory, and reading the segment into the allocated main memory. The segment must be active before it can be swapped into core, and the required main memory space must be available. Three conditions can be encountered during the invocation of SWAP_IN. The segment can already be located in global memory, the segment can already be located in one or more local memories, or the segment may only reside in secondary storage.

If the segment is not in local or global memory, local memory is allocated, the segment is read into the allocated memory, and the appropriate entries are made in the MMU image, the L_AST and the G_AST. If the segment is already in global memory, it can be assumed that the segment is shared and writable. In this case the only required actions are to update the G_AST and L_AST. The No_Active_In_Memory field of the G_AST entry is incremented, and the MMU image is updated to reflect the swapped in segment's core address and attributes.

If the segment already resides in one or more local memories, it must be determined if the segment is "shared" and

"writable". A segment is "shared" if it exists in more than one local memory. A segment is "writable" if one process has write access to that segment. If the segment is not shared or not writable and in local memory, the appropriate entries are updated in the MMU image, the L_AST, and the G_AST. If the segment does not reside in local memory, the required amount of local memory is allocated, the segment is read into the allocated memory, and the appropriate entries are made in the MMU image, the L_AST, and the G_AST.

If the segment is shared, writable, and in local memory, the segment must be moved to global memory. If the segment is not in the memory manager's local memory, it signals another memory manager to move the segment to global memory. After the segment is moved to global memory, the memory manager signals all of the connected memory manager's to update their L_AST and MMU data bases. When all local data bases are current, the memory manager updates the G_AST and returns a success code of seg_activated.

The pseudo-code for SWAP_IN PROCEDURE is presented in Figure 25. The arguments passed to SWAP_IN are the G_AST_INDEX of the segment to be moved in, the process' DBR_#, and the access authorized. SWAP_IN will convert the segment size from bytes to blocks, and verify that the process' core will not be exceeded. If the virtual core will be exceeded, a success code of "core_space_exceeded" will be returned. If write access is permitted, the writable bit is

set. Checks are then performed to determine the segment's storage location (local or global), and the appropriate action is taken.

```
PROCEDURE (INDEX
                           WORD, DBR_#
SWAP IN
                 ACCESS_AUTH
                             BYTE)
    RETURNS (SUCCESS CODE
                           BYTE)
          L_INDEX
                   WORD,
                            BLKS
                                   WORD
    ENTRY
    BLKS := CALCULATE_NO._OF_BLKS (G_AST[INDEX].SIZE)
    SUCCESS_CODE := CHECK_MAX_LINEAR_CORE (BLKS)
        SUCCESS CODE = VIRTUAL_LINEAR_CORE_FULL
                                                   THEN
        RETURN
    FI
    G_AST[INDEX].NO_SEGMENTS_IN_MEMORY += 1
    IF
         ACCESS AUTH = WRITE
                                THEN
        G_AST[INDEX].FLAG_BITS := WRITABLE_BIT_SET
    FI
  ! Determine if segment can be put in local memory !
    IF G_AST[INDEX].FLAG_BITS AND WRITABLE_MASK = 0
    ORIF G_AST[INDEX].NO_ACTIVE_IN_MEMORY <= 1 THEN
      ! Determine if already in local memory !
        CHECK_LOCAL_MEMORY (L_AST_INDEX)
            NOT_IN_LOCAL_MEMORY
                                  THEN
            ALLOCATE_LOCAL_MEMORY
                                     (BLKS)
            READ_SEGMENT (PAGE_TABLE_LOC, BASE_ADDR)
L_AST[L_INDEX] := BASE_ADDR
        FI
    ELSE
            NOT_IN_GLOBAL MEMORY
        IF
                                   THEN
            UPDATE_MMU
            UPDATE_L_AST
            RETURN
        ELSE
            ALLOCATE_GLOBAL_MEMORY (BLKS)
            IF IN LOCAL MEMORY
                                  THEN
              MOVE_TO_GLOBAL (L_INDEX, BASE_ADDR, SIZE)
            ELSE
              SIGNAL_OTHER_MEMORY_MANAGERS (INDEX, BASE_ADDR)
            FΙ
        FI
    FI
    UPDATE_MMU_IMAGE (DBR_#,SEG_#,BASE_ADDR,ACCESS,BLKS)
    UPDATE_L_AST_ACCESS (L_INDEX, ACCESS, DBR #)
    SUCCESS_CODE := SWAPPED IN
END
       SWAP IN
```

Figure 25: Swap_In Pseudo-code

6. Swap a Segment Out

SWAP_OUT is invoked when a user desires to move a segment out of core. A segment is swapped out of core by obtaining its secondary storage location, writing the segment to that location (if required), and deallocating the main memory used. The decision to write the segment is determined by the G_AST written bit. This bit is set whenever the segment has been modified. The segment to be swapped out can be in one of two states: the segment can be in local memory, or the segment can be in global memory.

If one process has the segment in local memory and the written bit is set, the segment is written into secondary storage and the local memory is deallocated. If the written bit is not set, the local memory need only be deallocated. If more than one process has the segment in the same local memory, the segment remains in core. The appropriate MMU image is updated to reflect the segments deletion and the G_AST No_Active_In_Memory field is decremented.

All segments in global memory are shared and writable. If a process requests the segment to be swapped out, the segment remains in memory. The MMU image is updated to reflect the segment's deletion, and the G_AST No_Active_In_Memory field is decremented. If the No_Active_In_Memory indicates that one process has the segment in core, its memory manager is signalled to move the segment to local memory.

The pseudo-code for SWAP OUT PROCEDURE is presented in Figure 26. The arguments passed to SWAP_OUT are the DBR_# of the signalling process, and the G_AST_INDEX of the segment to be removed. The return parameter is a success code. SWAP_OUT removes the segment from the process's core, deletes the segment from its MMU image, and decrements the No_Active_In_Memory field. If the segment can be removed from memory, it is determined which memory can be deallocated. If the segment has been modified, it is written back to secondary storage and the appropriate memory deallocated. If the segment has not been modified, the appropriate memory is deallocated. If after the deletion one process has the segment in global memory, its memory manager need only be signalled to move the segment to local memory. SWAP_OUT successfully completes, it returns a success code of "swapped out".

```
SWAP_OUT PROCEDURE (DBR_# BYTE, INDEX
                                         WORD)
    RETURNS (SUCCESS CODE BYTE)
    ENTRY
    BLKS := G_AST[INDEX].SIZE / BLK SIZE
   FREE_PROCESS_LINEAR_CORE (BLKS)
   DELETE_MMU_ENTRY (DBR_#, SEG #)
   G AST[INDEX]. NO SEGMENTS IN MEMORY -= 1
  ! Determine if segment has been written into !
   IF MMU_IMAGE(DBR_#].SDR(SEG_#].ATTRIBUTES=WRITTEN THEN
      ! If segment has been written into, update G_AST !
        G_AST[INDEX].FLAG_BITS := WRITTEN
   FI
  ! Determine if segment is in global memory !
   IF G AST[INDEX].GLOBAL ADDR <> NULL THEN
        IF G_AST[INDEX].NO_SEGMENTS_IN_MEMORY = 0
        ANDIF G_AST[INDEX].FLAG_BITS = WRITTEN
            WRITE SEG (PAGE TABLE LOC, MEMORY ADDR)
            FREE_LOCAL_BIT_MAP (MEMORY_ADDR, BLKS)
        ELSE
            IF G AST[INDEX].NO ACTIVE IN MEMORY = 0
                FREE_LOCAL_BIT_MAP (MEMORY_ADDR, BLKS)
            ΡI
        ΡI
   ELSE
            ! If not in global memory !
        IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0
        ANDIF G_AST[INDEX].FLAG_BITS = WRITTEN
            WRITE SEG (PAGE TABLE LOC, GLOBAL ADDR)
            FREE GLOBAL_BIT_MAP (GLOBAL_ADDR, BLKS)
            IF G_AST(INDEX).NO_ACTIVE_IN_MEMORY = 0 THEN
                FREE GLOBAL BIT MAP (GLOBAL ADDR, BLKS)
            FI
       FI
   FI
    SUCCESS CODE := SWAPPED_OUT
END
      SWAP_OUT
```

Figure 26: Swap_Out Pseudo-code

7. Deactivate All Sequents

DEACTIVATE_ALL is invoked when it becomes necessary to remove a segment from every process address space. Each process is checked to determine if the segment is active. If a process has the segment active, it is deactivated from its address space. The pseudo code for Deactivate_all is illustrated in Figure 27. The parameters passed to Deactivate_all are the deactivated segment's G_AST index and the L_AST index. The L_AST is searched by DBR_# to determine which process has the segment active. If the check reveals that the segment is active, it is deactivated by calling Deactivate. If the segment was successfully deactivated from all processes, a success_code of valid is returned.

```
DEACTIVATE_ALL PROCEDURE (INDEX WORD, L INDEX WORD)
    RETURNS (SUCCESS CODE BYTE)
    ENTRY
    LOCAL
         I
             BYTE
       I := 0
       DO
          IF I = MAX_DBR_# THEN
            EXIT
          FT
          IF
            L_AST[L_INDEX]. SEGMENT NO/ACCESS AUTH[I]
                 <> ZERO
                         THEN
             SUCCESS CODE := DEACTIVATE (I, INDEX)
                SUCCESS_CODE <> SEG_DEACTIVATED THEN
                RETURN
             FI
         FI
         I += 1
      OD
       SUCCESS CODE := VALID
END DEACTIVATE ALL
```

Figure 27: Deactivate All Pseudo-code

8. Move a Segment to Global Memory

MOVE_TO_GLOBAL is invoked when it becomes necessary to move a segment from local to global memory. If a segment resides in one or more local memories, and a process with write access swaps that segment into core, or if a segment resides in in local memory (with write access) and another process with read access requests the segment swapped in, the segment is moved from a local to global memory to avoid a secondary storage access. If the segment resides in the running memory manager's local memory, it will affect the

manager of a connected processor to affect the transfer. Figure 28 illustrates the pseudo-code for MOVE_TO_GLOBAL. Once the segment has been moved to global memory, the signalled memory manager will update the MMU images for all connected processes, and deallocate the freed local memory. A success code of completed will be returned to the signalling memory manager. The parameters passed to the memory manager are the segment's L_AST index the global memory address of the move, and the size of the segment. This information is passed because the G_AST is locked during this request.

```
MOVE_TO_GLOBAL PROCEDURE (L_INDEX WORD, GLOBAL_ADDR WORD, SIZE WORD)

RETURNS (SUCCESS_CODE BYTE)
ENTRY

! Move segment from local memory to global memory!
DO_MEMORY_MOVE (MEMORY_ADDR, GLOBAL_ADDR)
L_AST[INDEX]. MEMORY_ADDR := AVAILABLE
! Update the MMU image to reflect new address!
DO FOR_ALL_DBR'S

IF L_AST[L_INDEX]. SEGMENT_NO/ACCESS_AUTH <> O ANDIF

MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN

MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR:=GLOBAL_ADDR

FI

OD
SUCCESS_CODE := VALID
END MOVE_TO_GLOBAL
```

Figure 28: Move To Global Pseudo-code

9. Move a Sequent to Local Memory

MOVE_TO_LOCAL is invoked when it becomes necessary to move a segment from global to local memory. This occurs when one of two processes which hold a segment in global memory swaps the segment out. The segment is moved from global memory to the local memory of the remaining process. Figure 29 illustrates the pseudo-code for MOVE_TO_LOCAL. The parameters passed to the memory manager are the segment's L_AST index, the global address of the segment, and the size of the segment. The return parameter is a success code. The MMU images of the signalled process are updated after the move has been made, and the global memory is deallocated.

```
MOVE_TO_LOCAL PROCEDURE (L_INDEX WORD, GLOBAL_ADDR WORD,
                           SIZE WORD
    RETURNS (SUCCESS_CODE BYTE)
    ENTRY
    BLKS := SIZE / BLK_SIZE
    BASE_ADDRESS := ALLOCATE_LOCAL_MEMORY (BLKS)
  ! Move from global to local memory !
    MEMORY_MOVE (GLOBAL_ADDR, BASE_ADDRESS, SIZE)
    L_AST[L_INDEX]. MEMORY_ADDR := BASE_ADDRESS
    DO FOR ALL DER'S
      IF LAST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 ANDIP
      MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN
        MMU_IMAGE[ DBR_# ].SDR[ SEG_# ].BASE_ADDR:=BASE_ADDRESS
      FI
   OD
   SUCCESS_CODE := VALID
END
   MOVE_TO_LOCAL
```

Figure 29: Move To Local Pseudo-code

10. Update the MMU Image

UPDATE is invoked following a MOVE_TO_GLOBAL operation. After a segment has been moved from local memory to global memory, it is necessary to signal the memory managers of all connected processors to update their MMU images and L_AST with the current location of the segment. They must also deallocate the moved segment's local memory. Figure 30 illustrates the pseudo-code of UPDATE. The parameters passed to the memory manager are the segment's L_AST index, the new global address for the segment, and the size of the segment. The return parameter is a success code.

```
UPDATE
        PROCEDURE (L_INDEX WORD, GLOBAL ADDR WORD,
                    SIZE
                         WORD
    RETURNS (SUCCESS_CODE
                          BYTE)
    ENTRY
       FOR_ALL_DBR*S
    DO
         L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 ANDIF
      MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL_THEN
         MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR :=
                                          GLOBAL ADDR
      PI
    OD
    BLKS := SIZE / BLK_SIZE
    FREE_LOCAL_BIT_MAP (MEMORY ADDR.BLKS)
    L_AST[L_INDEX]. MEMORY_ADDR := ACTIVE
    SUCCESS_CODE := VALID
END UPDATE
```

Figure 30: Update Pseudo-code

E. SUMMARY

In this chapter the detailed design of the memory manager process has been presented. The purpose of the memory manager was outlined, followed by a detailed discussion of the memory manager's data bases. The design presented has identified ten basic functions for the memory manager. The success codes returned by the memory manager are presented in Figure 31.

This design has assumed that the kernel level inter-process synchronization primitives will be Saltzer's signal and wait primitives [14]. This fact dominated the design decision to lock the G_AST in the user's process before it signals the memory manager. In a multi-processor environment, the possibility of a deadly embrace exists if the memory manager processes lock the G_AST. Should follow on work implement eventcounts and sequencers as kernel level synchronization primitives, the locking of the G_AST and memory manager synchronization will need to be readdressed.

SYSTEM WIDE

INVALID
SWAPPED_IN
SWAPPED_OUT
SEG_ACTIVATED
SEG_DEACTIVATED
SEG_CREATED
SEG_DELETED
VIRTUAL_CORE_FULL
DUPLICATE_ENTRY
READ_ERROR
WRITE_ERROR
DRIVE_NOT_READY

KERNEL LOCAL

LEAF_SEGMENT_EXISTS

NO_LEAF_EXISTS
ALIAS_DOES_NOT_EXIST

NO_CHILD_TO_DELETE
G_AST_FULL
L_AST_FULL
LOCAL_MEMORY_FULL
GLOBAL_MEMORY_FULL
SECONDARY_STORAGE_FULL

MEMORY MANAGER LOCAL

VALID
INVALID
FOUND
NOT_FOUND
IN_LOCAL_MEMORY
NOT_IN_LOCAL_MEMORY
! + DISK ERRORS!

Figure 31: Success Codes

Chapter XII

STATUS OF RESEARCH

A. CONCLUSIONS

The memory manager design utilized state of the art software techniques and hardware devices. The design was developed based upon ZILOG'S Z8001 sixteen bit segmented microprocessor used in conjunction with the Z8010 Memory Management Unit [23]. A microprocessor which supports segmentation is required to provide access control of the stored data. The actual implementation of the selected thread was conducted upon the Z8002 non-segmented microprocessor without the Z8010 MMU.

while information security requires that the microprocessor support segmentation, the memory manager was developed to be configuration independent. The design will support a multi-processor environment, and can be easily implemented upon any microprocessor or secondary storage device. The loop free modular design facilitates any required expansion or modification.

Global bus contention is minimized by the memory manager. Segments are stored in global memory only if they are shared and writable. Secondary storage is accessed only if

the segment does not currently reside in global memory or some local memory. The controlled sharing of segments optimizes main memory usage.

The storage of the alias tables in secondary storage supports the recreation of user file hierarchies following a system crash. The aliasing scheme used to address segments supports system security by not allowing the segment's memory location or unique identification to leave the memory manager.

The design of the distributed kernel was clarified by assigning the MMU image management to the memory manager. The transfer of responsibility for memory allocation and deallocation from the supervisor to the memory manager provides support for dynamic memory management.

In conclusion, the memory manager process will securely manage segments in a multi-processor environment. The process is efficient, and is configuration independent. The primitives provided by the memory manager will support the construction of any desired supervisor/user process built upon the kernel.

B. FOLLOW ON WORK

There are several possible areas in the SASS design that can be looked into for continued research. The complete implementation of the memory manager design (refine and optimize the current PLZ/SYS code) is one possibility. Other possibilities include the implementation of dynamic memory management, and modifying the interface of the memory manager with the distributed kernel using eventcounts and sequencers for inter-process communication.

The implementation of the supervisor has not been addressed to date. Areas of research include the implmentation of the file manager and input/output processes, and the complete design and implementation of the user-host protocols. The implementation of the gatekeeper, and system initialization are other possible research areas. Dynamic process creation and deletion, and the introduction of multi-level hosts could also prove interesting.

PART D

AN IMPLEMENTATION OF MULTIPROGRAMMING AND PROCESS MANAGEMENT FOR A SECURITY KERNEL OPERATING SYSTEM

This section contains updated excerpts from a Naval Postgraduate School MS Thesis by S. L. Reitz [12]. The origins: of these excerpts are:

INTRODUCTION
IMPLEMENTATION
CONCLUSION

from Chapter I from Chapter IV from Chapter V

Minor changes have been made for integration into this repor

Chapter XIII

INTRODUCTION

The application of contemporary microprocessor technology to the design of large-scale multiple processor systems offers many potential benefits. The cost of high-power computer systems could be reduced drastically; fault tolerance in critical real-time systems could be improved; and computer services could be applied in areas where their use is not now cost effective. Designing such systems presents many formidable problems that have not been solved by the specialized single processor systems available today.

Specifically, there is an increasing demand for computer systems that provide protected storage and controlled access for sensitive information to be shared among a wide range of users. Data controlled by the Privacy Act, classified Department of Defence (DoD) information, and the transactions of financial institutions are but a few of the areas which require protection for multiple levels of sensitive information. Multiple processor systems which share data are well suited to providing such services - if the data security problem can be solved.

A solution to these problems - a multiprocessor system design with verifiable information security - is offered in

a family of secure, distributed multi-microprocessor operating systems designed by O'Connell and Richardson [7]. A subset of this family, the Secure Archival Storage System (SASS) [9,5], has been selected as a testbed for the general design. SASS will provide consolidated file storage for a network of possibly dissimilar "host" computers. The system will provide controlled, shared access to multiple levels of sensitive information (Figure 32).

This thesis presents an implementation of a basic monitor for the O'Connell-Richardson family of operating systems. The monitor provides multiprogramming and process management functions specifically addressed to the control of physical processor resources of SASS. Concurrent thesis work [7] is developing a detailed design for a security kernel process, the Memory Manager, which will manage SASS memory resources.

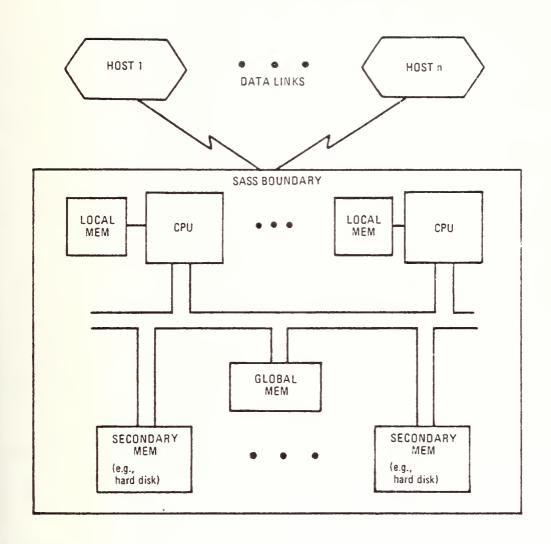


Figure 32: SASS SYSTEM

Chapter XIV

IMPLEMENTATION

Implementation of the distributed kernel was simplified by the hierarchical structure of the design for it permitted methodical bottom-up construction of a series of extended machines. This approach was particularly useful in this implementation since the bare machine, the Z8000 Developmental Module, was provided with only a small amount of software support.

A. DEVELOPMENTAL SUPPORT

A Zilog MCZ Developmental System provided support in developing Z8000 machine code. It provided floppy disk file management, a text editor, a linker and a loader that created an image of each Z8000 load module.

A Z8000 Developmental Module (DM) provided the necessary hardware support for operation of a Z8002 non-segmented microprocessor and 16K words (32K bytes) of dynamic RAM. It included a clock, a USART, serial and parallel I/O support, and a 2K PROM monitor.

The monitor provided access to processor registers and memory, single step and break point functions, basic I/O functions, and a download/upload capability with the MCZ system.

Since a segmented version of the processor was not available for system development, segmentation hardware was simulated in software as an MMU image (see Figure 33). Although this data structure did not provide the hardware support (traps) required to protect segments of the kernel domain, it preserved the general structure of the design.

 seg V	OFFSET High byte Low byte	ATTRIBUTES Size Attributes
	1	(

Figure 33: MMU_IMAGE

B. INNER TRAFFIC CONTROLLER

The Inner Traffic Controller runs on the bare machine to create a virtual environment for the remainder of the system. Only this module is dependent on the physical processor configuration of the system. All higher levels see only a set of running virtual processors. A kernel data base, the Virtual Processor Table is used by the Inner Traffic Controller to create the virtual environment of this first level extended machine. A source listing of the Inner Traffic Controller module is contained in Appendix G.

1. Virtual Processor Table (VPT)

The VPT is a data structure of arrays and records that maintains the data used by the Inner Traffic Controller to multiplex virtual processors on a real processor and to create the extended instruction set that controls virtual processor operation (see Figure 34). There is one table for each physical processor in the system. Since this implementation was for a uniprocessor system (the Z8000 DM), only one table was necessary.

The table contains a LOCK which supports an exclusion mechanism for a multiprocessor system. It was provided in this implementation only to preserve the generality of the design.

The Descriptor Base Register (DBR) binds a process to a virtual processor. The DBR points to an MMU_IMAGE contain-

LOCK
RUNNING_LIST
READY_LIST
FREE_LIST

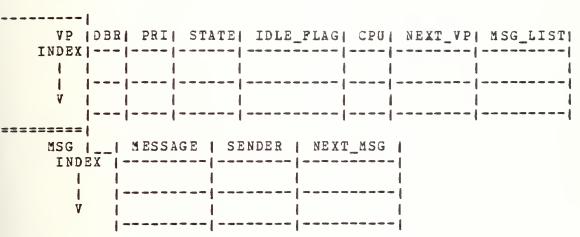


Figure 34: Virtual Processor Table

ing the list of descriptors for segments in the process address space.

A virtual processor (VP) can be in one of three states: running, ready, and waiting (Figure 35). A running VP is currently scheduled on a real processor. A ready VP is ready to be scheduled when selected by the level-1 schedul-

ing algorithm. A waiting VP is awaiting a message from some other VP to place it in the ready list. In the meantime it is not in contention for the real processor.

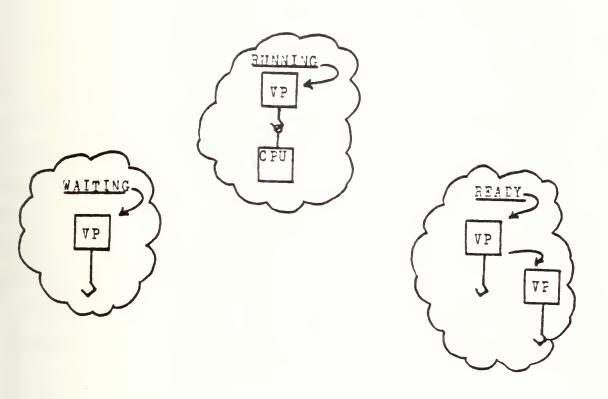


Figure 35: Virtual Processor States

Level-1 Scheduling

Virtual processor state changes are initiated by the inter-virtual-processor communication mechanisms, SIGNAL and These level-1 instructions implement the scheduling WAIT. policy by determining what virtual processor to bind to the real processor. The actual binding and unbinding is performed by a Processor switching mechanism called SWAP_DBR [14]. Processor switching implies that somehow the execution point and address space of a new process are acquired by the processor. Care must be taken to insure that the old process is saved and the new process loaded in an orderly manner. A solution to this problem, suggested by Saltzer: [14], is to design the switching mechanism so that it is a common procedure having the same segment number in every address space.

In this implementation a processor register (R14) was reserved within the switching mechanism for use as a DBR. Processor switching was performed by saving the old execution point (i.e., processor registers and flag controlword), loading the new DBR and then loading the new execution point. The processor switch occurs at the instant the DBR is changed (see Figure 36). Because the switching procedure is distributed in the same numbered segment in alladdress spaces, the "next" instruction at the instant of the switch will have the same offset no matter what address

space the processor is in. This is the key to the proper operation of SWAP_DBR.

To convert this switching mechanism to segmented hard-ware it is necessary merely to replace SWAP_DBR with special I/O block-move instructions that save the contents of the MMU in the appropriate MMU_IMAGE and load the contents of the new MMU_IMAGE into the MMU.

a. Getwork

SWAP_DBR is contained within an internal Inner Traffic Controller procedure called GETWORK. In addition to multiplexing virtual processors on the CPU, GETWORK interprets the virtual processor status flags, IDLE and PREEMPT, and modifies VP scheduling accordingly in an attempt to keep the CPU busy doing useful work.

There are actually two classes of idle processes within the system. One class belongs to the Traffic Controller. Conceptually there is a ready level-2 idle process for each virtual processor available to the Traffic Controller for scheduling. When a running process blocks itself, the Traffic Controller schedules the first ready process. This will be an idle process if no supervisor processes are in the ready list.

The second class of idle process exists in the kernel.

The kernel Idle process is permanently bound to the lowest priority virtual processor.

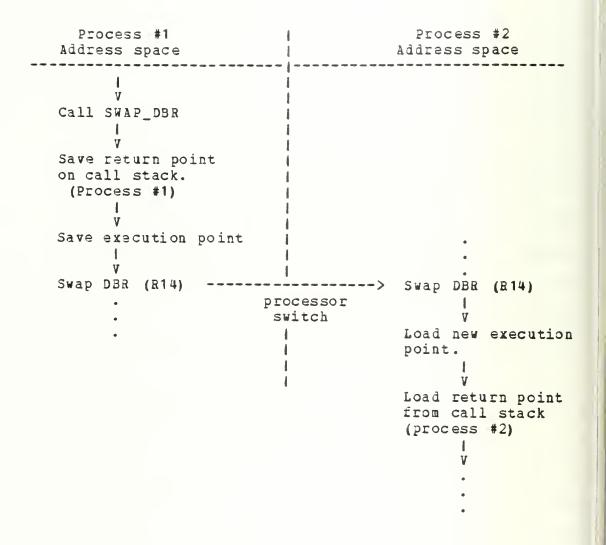


Figure 36: SWAP_DBR

The distinction is made between these classes because of the need to keep the CPU busy doing useful work whenever possible. There is no need for GETWORK to schedule a level-2 idle process that has been loaded on a virtual processor, because the idle process does no useful work. The virtual processor IDLE_FLAG indicates that a virtual processor has been loaded with a level-2 idle process. GETWORK will schedule this virtual processor only if the PREEMPT flag is also set. The PREEMPT flag is a signal from the Traffic Controller that a supervisor process is now ready to run.

When GETWORK can find no other ready virtual processors with IDLE and PREEMPT flags off, it will select the virtual processor permanently bound to the kernel Idle process.

Only then will the Idle process actually run on the CPU.

entry, resets the preempt interrupt return flag. (RO is reserved for this purpose within GETWORK.) The second, a hardware interrupt entry point, contains an interrupt handler which sets the preempt interrupt return flag. The DBR (R14) must also be set to the current value by any procedure that calls GETWORK in order to permit the SWAP_DBR portion of GETWORK to have access to the scheduled process's address space. Upon completion of the processor switch, GETWORK eximines the interrupt return flag to determine whether a normal return or an interrupt return is required.

The hardware interrupt entry point in GETWORK supports the technique used to initialize the system. Each process address space contains a kernel domain stack segment used by SWAP-DBR in GETWORK to save and restore VP states. same reason that SWAP-DBR is contained in a system wide segment number, the stack segment in each process address space will also have the same number (Segment #1 in this implementation). Each stack segment is initially created as though it's process had been previously preempted by a hardware interrupt. This greatly simplifies the initialization of processes at system generation time. The details of system initialization will be described later in this chapter. It is important to note here, however, that GETWORK must be able; to determine whether it was invoked by a hardware preempt: interrupt or by a normal call, before it can execute a return to the calling procedure. This is because a hardware interrupt causes three items to be placed on the system stack: the return location of the caller, the flag control word, and the interrupt identifier, whereas a normal call places only the return location on the stack. Therefore, in order to clean up the stack, GETWORK must execute an interrupt return (assembly instruction: IRET) if entry was via the hardware preempt handler (i.e., RO set). This instruction will pop the three items off the stack and return to the appropriate location. If the interrupt return flag, RO, is off, a normal return is executed.

During normal operation, SWAP-DBR manipulates process stacks to save the old VP state and load the new VP state.

This action proceeds as follows (Figure 37):

- 1. The Flag Control Word (FCW), the Stack Pointer (R15) and the preempt return flag (R0) are saved in the old VP's kernel stack.
- 2. The DBR (R14) is loaded with the new VP's DBR. This permits access to the address space of the new process.
- 3. The Flag Control Word (FCW), the Stack Pointer (R15) and the Interrupt Return Flag (R0), are loaded into the appropriate CPU registers.
- 4. RO is tested. If it is set, GETWORK will execute an interrupt return. If it is off, a normal return occurs.

By constructing GETWORK in this way, both system initialization and normal operations can be handled in the same way.

A high level GETWORK algorithm is given in Figure 38.

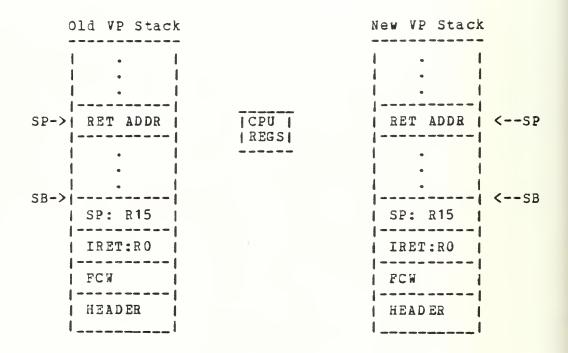


Figure 37: Kernel Stack Segments

```
GETWORK Procedure (DBR = R14)
 Begin
  Reset Interrupt Return Flag (RO)
  Skip hardware preempt handler
 Hardware Preempt Entry:
    Set DBR
    Save CPU registers
    Save supervisor stack pointer
    Set Interrupt Return Flag (RO)
  Get first ready VP
  Do while not Select
   If Idle flag is set then
    if Preempt flag is set then
     select
    else
     get next ready VP
    end if
   else
    select
   end if
  end do
  SWAP_DBR:
   Save old VP registers in stack segment
   Swap dbr (R14)
   Load new VP registers in stack segment
   If Interrupt Return Flag is set then
    unlock VPT
    simulate GATEKEEPER exit:
     Call TEST_VPREEMPT
     Restore supervvisor registers
     Restore supervvisor stack pointer
    Execute Interrupt Return (IRET)
   end if
   Execute normal return
end GETWORK
```

Figure 38: GETWORK

3. <u>Virtual Processor Instruction Set</u>

The heart of the SASS scheduling mechanism is the internal procedure, GETWORK. It provides a powerful internal primitive for use by the virtual processors and greatly simplifies the design of the virtual processor instruction set. Virtual processor instructions perform three types of functions: multiprogramming, process management and virtual interrupts.

tion between virtual processors. They multiplex virtual processors on a CPU to provide multiprogramming. This implementation used a version of the signal and wait algorithms proposed by Saltzer [14]. In the SASS design each CPU is provided with a unique (fixed) set of virtual processors. The interaction among virtual processors is a result of multiprogramming them on the real processor. Only one virtual processor is able to access the VPT at a time because of the use of the VPT LOCK (SPIN_LOCK) to provide mutual exclusion. Therefore race and deadlock conditions will not develop and the signal pending switch used by Saltzer is not necessary.

This implementation also included message passing mechanism not provided by Saltzer. The message slots available for use by virtual processors are initially contained in a queue pointed to by FREE-LIST. When a message is sent from one VP to another, a message slot is removed from the free

list and placed in a FIFO message queue belonging to the VP receiving the message. The head of each VP's message queue is pointed to by MSG-LIST. Each message slot contains a message, the ID of the sender, and a pointer to the next message in the list (either the free list or the VP message list.

IDLE and SWAP_VDBR provide the Traffic Controller with a means of scheduling processes on the running VP.

SET_VPREEMPT and TEST_VPREEMPT install a virtual interrupt mechanism in each virtual processor. When the Traffic
Controller determines that a virtual processor should give
up its process because a higher priority process is now
ready, it sets the PREEMPT flag in that VP. Then, even if
an idle process is loaded on the VP, it will be scheduled
and will be loaded with the first ready process.
Test_VPreempt is a virtual interrupt unmasking mechanism
which forces a process to examine the preempt flag each time
it exists from the kernel.

a. Wait

WAIT provides a means for a virtual processor to move itself from the running state to the waiting state when it has no more work to do. It is invoked only for system events that are always of short duration. It is supported by three internal Procedures.

SPIN_LOCK enables the running VP to gain control of the Virtual Processor Table. This procedure is only necessary in a multiprocessor environment. The running VP will have to wait only a short amount of time to gain control of the VPT. SPIN_LOCK returns when the VP has locked the VPT.

GETWORK loads the first eligible virtual processor of the ready list on the real processor. Before this procedure is invoked, the running VP is placed in the ready state. Both ready and running VP's are members of a FIFO queue. GETWORK selects the first VP in this ready list, loads it on the CPU, and places it in the running state. When GETWORK returns, the first VP of the queue will always be running; and the second will be the first VP in the ready queue.

GET_FIRST_MESSAGE returns the first message of the message list (also managed as a FIFO queue) associated with the running VP. The action taken by WAIT is as follows:

WAIT Procedure (Returns: Msg. Sender_ID)
Begin

Lock VPT (call SPIN_LOCK)

If message list empty (i.e., no work) Then

Move VP from Running to Waiting state

Schedule first eligible Ready VP (call GETWORK)

end if

(NOTE: process suspended here until

it receives a signal and is selected by GETWORK.)

Get first message from message list

(call GET_FIRST_MSG)

Unlock VPT

Return

end WAIT

If the running virtual processor calls WAIT and there is a message in its message list (placed there when another VP signaled it) it will get the message and continue to run. If the message list is empty it will place itself in the wait state, schedule the first ready virtual processor, and move it to the running state. The virtual processor will remain in the waiting state until another running VP sends it a message (via SIGNAL). It will then move to the ready list. Finally it will be selected by GETWORK, the next instructions of WAIT will be executed, it will receive the message for which it was waiting, and it will return to the caller.

b. Signal

Messages are passed between virtual processors by the instruction, SIGNAL, which uses four internal procedures, SPIN_LOCK, ENTER_MSG_LIST, MAKE_READY, and GETWORK.

SPIN_LOCK, as explained above insures that only one virtual processor has control of the Virtual Processor Table at a time.

ENTER_MSG_LIST manages a FIFO message queue for each virtual Processor and for free messages. This queue is of fixed maximum length because of the implementation decision to restrict the use of SIGNAL. A running VP can send no more than one message (SIGNAL) before it receives a reply (i.e., WAIT's for a message). Therefore if there are N virtual processors per real processors, the message queue length, L, is:

L = N - 1

MAKE_READY manages the virtual processor ready queue. If a message is sent to a VP in the waiting state, MAKE_READY wakes it up (it places it in the ready state) and enters it in the ready list. If a running VP signals a waiting VP of higher priority, it will place itself back in the ready state and the higher priority VP will be selected. The action taken by signal is as follows:

SIGNAL Procedure (Message, Destination_VP)
Begin

Lock VPT (call SPIN_LOCK)

Send message (call ENTER_MSG_LIST)

If signaled VP is waiting Then

Wake it up and make it ready

(call MAKE_READY)

end if

Put running VP in ready state.

Schedule first elgible ready VP

(call GETWORK)

Unlock VPT

Return (Success_code)

End SIGNAL

C. SWAP_VDBR

SWAP_VDBR contains the same processor switching mechanism used in SWAP_DBR, but applies it to a virtual processor rather than a real processor. Switching is quite simple in this virtual environment because both processor execution point and address space are defined by the Descriptor Base

Register. SWAP_VDBR is invoked by the Traffic Controller to load a new process on a virtual processor in support of level-2 scheduling. It uses GETWORK to control the associated level-1 scheduling. The action taken by SWAP_VDBR is:

SWAP_VDBR Procedure (New_DBR)

Begin

Lock VPT (call SPIN_LOCK)

Load running VP with New_DBR

Place running VP in ready state

Schedule first eligible ready VP

(call GETWORK)

Unlock VPT

Return

End SWAP_VDBR

In this implementation one restriction is placed upon the use of this instruction. If a virtual processor's message list contains at least one message, it can not give up its current DBR. This problem is avoided as the natural result of using SIGNAL and WAIT only for system events, and "masking" preempts within the kernel. If this were permitted, the messages would lose their context. (The messages in a VP_MSG_LIST are actually intended for the process loaded on the VP.)

d. IDLE

The IDLE instruction loads the Idle DBR on the running virtual processor. Only virtual processors in contention for process scheduling will be loaded by this instruction. (The Traffic Controller is not even aware of virtual processors permanently bound to kernel processes.)

IDLE has the same scheduling effect as SWAP VDBR, but it also sets the IDLE_FLAG on the scheduled VP. The distinction is made between the two cases because, although the Traffic Controller must schedule an Idle process on the VP if there are no other ready processes, the Inner Traffic Controller does not wish to schedule an Idle VP if there is an alternative. This would be a waste of physical processor resources. The setting of the IDLE_FLAG by the Traffic Controller aids the Inner Traffic Controller in making this scheduling decision. Logically, there is an idle process for each VP; actually the same address space (DBR) is used for all idle processes for the same CPU, since only one will run at a time. As previously explained, virtual processors loaded by this instruction will be selected by GETWORK only to give the Idle process away for a new process in response to a virtual preempt interrupt. The action of IDLE is:

IDLE Procedure

Begin

Lock VPT (call SPIN_LOCK)

Load running VP with Idle DBR

Set VP's IDLE_FLAG

Place running VP in ready state

Schedule first elgible ready VP

(call GETWORK)

Unlock VPT

Return

End IDLE

e. SET_VPREEMPT

SET_VPREEMPT sets the preempt interrupt flag on a specified virtual processor. This forces the virtual processor into level-1 scheduling contention, even if it is loaded with an Idle process. The instruction retrieves an idle

virtual processor in the same way a hardware preempt retrieves an idle CPU by forcing the VP to be selected by GETWORK. The only difference between the two cases is the entry point used in GETWORK. The action of SET_VPREEMPT is:

SET_VPREEMPT Procedure (VP)

Begin

Set VP's PREEMPT flag

If VP belongs to another CPU Then

send hardware interrupt

end if

Return

End SET_VPREEMPT

Since the action is a safe sequence, no deadlocks or race conditions will arise and no lock is required on the VPT.

f. TEST_VPREEMPT

within the kernel of a multiprocessor system all process interrupts (which excludes system I/O interrupts) are masked. If process interaction results in a virtual preempt being sent to the running virtual processor by another CPU, it will not be handled since GETWORK has already been invoked. TEST_VPREEMPT provides a virtual preempt interrupt unmasking mechanism.

TEST VPREEMPT mimics the action of a physical CPU when interrupts are unmasked. It forces the process execution point back down into the kernel each time the process attempts to leave the kernel domain, where the preempt flag of the running VP is examined. If the flag is off, TEST_VPREEMPT returns and the execution point exits through the Gatekeeper into the supervisor domain of the process address space as described above. However, if the PREEMPT flag is on, the TEST_VPREEMPT executes a virtual interrupt handler located in the Traffic Controller. This jump from the Inner Traffic Controller to the Traffic Controller (TC_PREEMPT_HANDLER) is a close parallel to the action of a CPU in response to a hardware interrupt, that is a jump to an interrupt handler. The Traffic Controller Preempt Handler forces level-2 and level-1 scheduling to proceed in the normal manner. The preempt handler forces the Traffic Controller to examine the APT and to apply the level-2 scheduling algorithm, TC_GETWORK. If the APT has been changed since the last invocation of this scheduler, it will be reflected in the scheduling selections. Eventually, when the running VP's preempt flag is tested and found to be reset, TEST_VPREEMPT will return to the Gatekeeper where the process execution point will finally make a normal exit into its supervisor domain. TEST_VPREEMPT performs the following action:

TEST_VPREEMPT Procedure

Begin

Do while running VP's PREEMPT flag is set

Reset PREEMPT flag

Call preempt handler

(call TC_PREEMPT_HANDLER)

End do

Return

End TEST_VPREEMPT

C. TRAFFIC CONTROLLER

The Traffic Controller runs in a virtual environment created by the Inner Traffic Controller. It sees a set of running virtual processor instructions: SWAP_VDBR, IDLE, SET_VPREEMPT, and RUNNING_VP, and provides a scheduler, TC_GETWORK, which multiplexes processes on virtual processors in response to process interaction. It also creates a level-2 instruction set: ADVANCE, AWAIT, and PROCESS_CLASS, which is available for use by higher levels of the design. The Traffic Controller uses a global data base, the ACTIVE PROCESS TABLE to support its operation.

1. Active Process Table (APT)

The Active Process Table is a system-wide kernel database containing entries for each supervisor process in SASS (Figure 39). It is indexed by active process ID. The structure of the APT closely parallels that of the Virtual Processor Table. It contains a LOCK to support the implementation of a mutual exclusion mechanism, a RUNNING_LIST, and a READY_LIST_HEAD. The Traffic Controller is only concerned with virtual processors that can be loaded with supervisor processes. Since two VP's are permanently bound to kernel processes (the Memory Manager and the Idle Process), they cannot be in contention for level-2 scheduling; the Traffic Controller is unaware of their existence; since there are a number of available virtual processors, the

RUNNING_LIST was implemented as an array indexed by VP_ID.

The READY_LIST_HEAD points to a FIFO queue that includes both running and ready processes. The running processes will be at the top of the ready list.

Because of their completely static nature, idle processes require no entries in the APT. Logically, there is an idle process at the end of the ready list for each VP available to the Traffic Controller. If the ready list is empty, TC_GETWORK loads one of these "virtual" idle processes by calling IDLE, and enters a reserved identifier, #IDLE, in the appropriate RUNNING_LIST entry. This identifier is the only data concerning idle processes that is contained in the APT. Idle process scheduling considerations are moved down to level-1, because the Inner Traffic Controller knows about physical processors, and can optimize CPU use by scheduling idle processes only when there is nothing else to do.

The subject access class, S_CLASS, provides each process with a label that is required by level-3 modules to enforce, the SASS non-discretionary security policy.

LOCK					
RUNNI	NG_LIS	PROCESS_ID			
1 1 1 b	ID				
	LIST_				
	DBR 	ACCESS_CLASS	STATE	NEXT_AP	EVENTCOUNT HANDLE INSTANCE COUNT
AP Index 1	 			 	

Figure 39: Active Process Table

2. Level-2 Scheduling

Above the Traffic Controller, SASS appears as a collection of processes in one of the three states: running, ready, or blocked. Running and ready states are analogous to the corresponding virtual processor states of the Inner Traffic Controller. However, because of the use of event-count synchronization mechanisms by the Traffic Controller, the blocked state has a slightly different connotation than the VP waiting state.

Blocked processes are waiting for the occurrence of a non-system event, e.g., the event occurrence may be signalled from the supervisor domain. When a specific event happens, all of the blocked processes that were awaiting that event are awakened and placed in the ready state. This broadcast feature of event occurrence is more powerful than the message passing mechanism of SIGNAL, which must be directed at a single recipient.

Just as SIGNAL and WAIT provide virtual processor multiplixing in level-1, the eventcount functions, ADVANCE and AWAIT, control process scheduling in level-2.

a. TC_GETWORK

Level-2 scheduling is implemented in the internal Traffic Controller procedure, TC_GZTWORK. This procedure is invoked by eventcount functions when a process state change

may have occurred. It loads the first ready process on the currently scheduled VP (i.e., the virtual processor that has been scheduled at level-1 and is currently executing on the CPU).

```
TC_GETWORK Procedure
 Begin
  VP_ID := RUNNING_VP
  Do while not end of ready list
   if process is running then
   get next ready process
   else
    RUNNING_LIST [VP_ID] := PROCESS_ID
    Process state := running
    SWAP_VDBR
   end if
  end do
If end of running list (no ready processes) Then
  RUNNING_LIST := #IDLE
 IDLE
end if
Return
End TC_GETWORK
```

b. TC_PREEMPT_HANDLER

Preempt interrupts are masked while a process is executing in the kernel domain. As the process leaves the kernel, the gatekeeper unmasks this virtual interrupt by invoking TEST_VPREEMPT. This instruction tests the scheduled VP's If this flag is off, the process returns to PREEMPT flag. the Gatekeeper and exits from the kernel; but if the flag is set, TEST_VPREEMPT calls the Traffic Controller's virtual preempt interrupt handler, TC_PREEMPT_HANDLER. This handler invokes IC_GETWORK, which re-evaluates level-2 scheduling. Eventually, when the schedulers have completed their functions, the handler will return control to the preempted process, which will return to te Gatekeeper for a normal exit. This sequence of events closely parallels the action of a hardware interrupt, but in the environment of a virtual processor rather than a CPU. The virtualization of interrupts provides the ability for one virtual processor to interrupt execution of another that may, or may not, be running on a CPU at that time. This is provided without disrupting the logical structure of the system. This capability is particularly useful in a multiprocessor environment where the target virtual processor may be executing on another CPU. Because these interrupts will be wirtualized, the operating system will retain control of the system. The action of the TC PREEMPT HANDLER is described in the procedure below.

TC_PREEMPT_HANDLER Procedure

Begin

Call WAIT_LOCK

VP_ID := RUNNING_VP

Process_ID := RUNNING LIST (VP_ID)

If process is not idle Then

Process state := ready

end if

Call TC_GETWORK

Call WAIT_UNLOCK

RETURN

WAIT_LJCK and WAIT_UNLOCK provide an exclusion mechanism which prevents simultaneous multiple use of the APT in a multiprocessor configuration. This mechanism invokes WAIT and SIGNAL of the Inner Traffic Controller.

End TC_PREEMPT_HANDLER

3. Eventcounts

An eventcount is a non-decreasing integer associated with a global object called an event [11]. The Event Manager, a level-3 module, controls access to event data when required and provides the Traffic Controller with a HANDLE, an INSTANCE, and a COUNT. The values for all eventcounts (and sequencers) are maintained at the Memory Manager level and are accessed by calls to the Memory Manager. The HANDLE provides the traffic controller with an event ID, associated with a particular segment. INSTANCE is a more specific definition of the event. For example, each SASS supervisor segment has two eventcounts associated with it, a INSTANCE 1 and a INSTANCE_2, that the supervisor uses keep track of read and write access to the segment [9]. Eventcounts provide information concerning system-wide events. manipulated by the Traffic Controller functions ADVANCE and AWAIT and by the Memory Manager functions, READ and TICKET. A proposed high level design for ADVANCE and AWAIT is provided by Reitz [12].

a. Advance

ADVANCE signals the occurrence of an event (e.g., a read access to a particular supervisor segment). The value of the eventcount is the number of ADVANCE operations that have been performed on it. When an event is advanced, the fact must be broadcast to all blocked processes awaiting it and

the process must be awakened and placed on the ready list. Some of the newly awakened processes may have a higher priority than some of the running processes. In this case a virtual preempt, SET_VPREEMPT (VP_ID), must be sent to the virtual processors loaded with these lower priority processes.

b. Await

When a process desired to block itself until a particular event occurs, it invokes AWAIT. This procedure returns to the calling process when a specified eventcount is reached. Its function is similar to WAIT.

c. Read

READ returns the current value of the eventcount. This is an Event Manager (level three) function. This module calls the Memory Manager module to obtain the eventcount value.

d. Ticket

TICKET provides a complete time-ordering of possibly concurrent events. It uses a non-decreasing integer, called a sequencer, which is also associated with each supervisor segment. As with READ, this is an Event Manager function that calls the Memory Manager to access the sequencer value. Each invocation of TICKET increments the value of the se-

quencer and returns it to the caller. Two different uses of ticket will return two different values, corresponding to the order in which the calls were made.

D. SYSTEM INITIALIZATION

Because the Inner Traffic Controller's scheduler, GETWORK, can accommodate both normal calls and hardware interrupt jumps, the problem of system initialization is not difficult.

When SASS is first started at level-1, the Idle VP is running and the memory manager VP, which has the highest priority, is the first ready virtual processor in the ready list. All VP's available to the Traffic Controller for level-2 schedling are ready. Their IDLE_FLAG's and PREEMPT flags are set.

At level-2, all VP's are loaded with idle processes and all supervisor processes are ready.

The kernel stack segment of each process is initialized to appear as if it had been saved by a hardware Preempt interrupt (Figure 40).

All CPU registers and the supervisor stack pointer are stored on the stack. R15 is reserved as the kernel stack point; R14 contains the DBR. All other registers can be used to pass initial parameters to the process. The order in which these registers appear on the stack supports the PLZ/ASM block-move instructions.

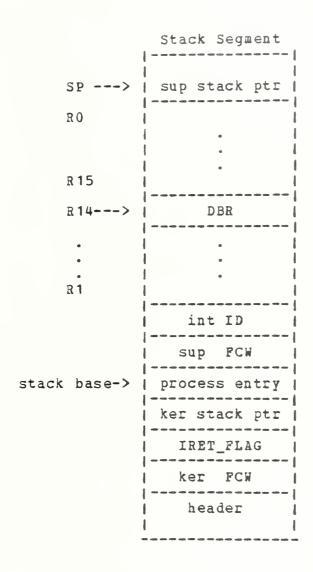


Figure 40: Initialized Stack

The status block contains the current value of the stack pointer, R15, and the preempt interrupt return flag. This flag is set to indicate that the process has been saved by a preempt interrupt. The first three items on the stack: the process entry point, the initial process flag control word, and an interrupt indentifier, are also initialized to support the action of a hardware interrupt.

To start-up the system, R14 (the DBR) is set to the Idle process DBR; the CPU Program counter is assigned the PREEMPT_ENTRY point in GETWORK; the CPU Flag Control Word (FCW) is initialized for the kernel domain: and the CPU is started. Because the Idle_VP is the lowest priority VP in the system, it will place itself back in the ready state and move the Memory Manager in the running state. The Memory Manager will execute an interrupt return because the interrupt return flag was set by system initialization. There will be no work for this kernel process so it will call WAIT to place itself in the waiting state. The next ready VP is idling, but since it's IDLE_FLAG and PREEMPT flag are set, GETWORK will select it. It too will execute an interrupt return, but because its PREEMPT flag is set, it will call TC_PREEMPT_HANDLER. This will cause the first ready process to be scheduled. Each time a supervisor process blocks itself, the next idle VP will be selected and the sequence will be repeated.

The action described above is in accord with normal operation of the system. The only unique features of initialization are the entry point (PREEMPT-ENTRY: in GETWORK) and the values in the initialized kernel stack.

The implementation presented in this thesis has been run on a Z8000 developmental module. System initialization has been tested and executes correctly. At the current level of implementation, no process multiplexing function is available. There is no provision for unlocking the APT after an initialized process has been loaded as a result, a call to the Traffic Controller (viz., ADVANCE or AWAIT). In a process multiplexed environment this would cause a system deadlock. Once the process left the kernel domain with a locked APT, no process would be able to unlock it. The Traffic Controller must handle this system initialization problem.

Chapter XV

CONCLUSION

The implementation presented in this thesis created a security kernel monitor that runs on the Z8000 Developmental Module. This monitor supports multiprogramming and process management in a distributed operating system. The process executes in a multiple virtual processor environment which is independent of the CPU configuration.

This monitor was designed specifically to support the Secure Archival Storage System (SASS) [2, 9, 5]. However, the implementation is based on a family of Operating Systems [7] designed with a primary goal of providing multilevel security of information. Although the monitor currently runs on a single microprocessor system, the implementation fully supports a multiprocessor design.

A. RECOMMENDATIONS

Because the Zilog MMU is not yet available for the Z8000 Developmental Module, it was necessary to simulate the segmentation hardware. As Reitz explained [12], this was accomplished by reserving a CPU register, R14, as a Descriptor Base Register (DBR) to provide a link to the loaded addresss space. When the MMU becomes available, this simulation must be removed. This can be done in two steps.

First, the addressing format must be translated to the segmented form. This requires no system redesign.

Second, the switching mechanism most be modified to accommodated to use the MMU. This can be done by modifying the SWAP_DBR portion of GETWORK to multiplex the MMU_IMAGE onto the MMU hardware and this can be accomplished by changing about a dozen lines of the existing code.

B. FOLLOW ON WORK

Although the monitor appears to execute correctly, it has not been rigorously tested. Before higher levels of the system are added, it is essential that the monitor be highly reliable. Therefore a formal test and evaluation plan should be developed.

An automated system generation and initialization mechanism is also required if the monitor to be is a useful tool in the development of higher levels of the design.

Once the monitor has been proven reliable and can be loaded easily, work on the implementation of the Memory Manager kernel process and the remainder of the kernel can continue.

PART E

IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STORAGE SYSTEM

This section contains excerpts from a Naval Postgraduate School MS Thesis by J. T. Wells [20]. The origins of these excerpts are:

INTEODUCTION from Chapter I
SEGMENT MANAGEMENT FUNCTIONS
SEGMENT MANAGER
NON-DISCRETIONARY SECURITY MODULE
MEMORY MANAGER
SUMMARY from Chapter II

SUMMARY from Chapter II
SEGMENT MANAGEMENT IMPLEMENTATION from Chapter III
CONCLUSIONS AND FOLLOW ON WORK from Chapter IV

Minor changes have been made for integration into this report.

Chapter XVI

INTRODUCTION

This thesis addresses the implementation of the segment management functions of an operating system known as the Secure Archival Storage System or SASS. This system, with full implementation, will provide: (1) multilevel secure access to information (files) stored in a "data warehouse" for a network of multiple host computers, and (2) controlled data sharing among authorized users. The correct performance of both of these features is directly dependent upon the proper implementation of the segment management functions addressed in this thesis. The issue of access to sensitive information is addressed by the Non-Discretionary Security Module, which mediates all non-discretionary access to in-Sharing of information is accomplished chiefly through the properties of segmentation, the SASS memory management scheme that is supported by the Memory Manager Module and the Segment Manager Module. The implementation of segment management for SASS is thus integral to the attainment of the two key goals that SASS was designed to achieve. This implementation addresses the Non-Discretionary Security, Distributed Memory Manager (the interface to the Memory Manager Process), and Segment Manager modules.

Chapter XVII

SEGMENT MANAGEMENT FUNCTIONS

A. SEGMENT MANAGER

1. Function

The Segment Manager is the focal point of the segment management function. Using the per-process Known Segment Table as its database and the Memory Manager and Non-Discretionary Security Module in strongly supportive roles, it is responsible for managing the segmented virtual memory for a process. Its role can be viewed as somewhat intermediary in nature (viz., between the Supervisor modules and the Memory Manager modules). The extended instruction set created in the Segment Manager includes the following instructions: CREATE_SEGMENT, DELETE_SEGMENT, MAKE_KNCWN, TERMINATE SM SWAP IN, and SM SWAP OUT (note that the names for SWAP IN and SWAP_OUT have been modified by preceding each with SM_; this is strictly for clarity because the Memory Manager also creates two instructions called SWAP_IN and SWAP_CUT). These instructions are invoked by the Supervisor domain of the process (viz., calls are made from the Supervisor domain via the Gatekeeper to the Segment Manager in the Kernel domain) to provide SASS support to the Host.

In general, when the Segment Manager receives these calls, it performs certain checks to ensure the validity and security compliance (when required) of the request (call). These checks are performed using its own database (the KST) and by calls to the Non-Discretionary Security Module (when required). The Segment Manager invokes one of six Memory Manager (more specifically, the Distributed Memory Manager Module) created instructions. These instructions include: MM_DELETE_ENTRY, MM_CREATE_ENTRY, MM ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. These invoked | instructions (procedures) in turn perform interprocess communciations with the non-distributed memory manager process (where actual memory management functions are accomplished). These interprocess invocations and returns are accomplished through the use of the IPC primitives Signal and Wait. The Segment Manager returns the required arguments to the Supervisor by value (as passed back to it by the Memory Manager and/or determined within itself). The Segment Manager performs actual segment number assignment when a segment is made known to a process' address space. It also performs any further database (KST) updating as may be required.

2. Database

The Known Segment Table (KST) is the database used to manage segments. The KST is described in its tabular form and PLZ/SYS structured representation in Figure 41. There

are several basic and pertinent facts to be noted of the
KST:

- 1. It is a process local database; that is, each process has its own KST.
- 2. The KST is indexed by segment number: each record of the KST consists of a set of fields (description information) regarding a particular segment.
- 3. Entering information into the fields of a segment is called "making a segment known". This simply refers to adding a segment to a process' address space (viz., making a segment accessible to a process).
- 4. In SASS, a correspondence exists between making a segment "known" and making a segment "active": i.e., when a segment is added to the address space of a process, this action results in an entry in the KST (making "known") by the Segment Manager and an entry in the Global Active Segment Table (G_AST) by the Memory Manager process (making it "active"). The G_AST will be described later in this chapter.

A proper description of the structure and fields of the KST is necessary at this point. Using the representation of the PLZ/SYS language structure, the KST is described as an array of records of fields of varying types. The fields are described separately below. Although the KST index is not in itself a field in the record, it does perform a rather significant role. The KST index is an integer closely related

to the segment number of the segment described in that KST entry (viz., it is the subscript into the array of records). This segment number also corresponds to the MMU descriptor register (number) for that segment.

The MM_Handle is the first field in a KST record. The MM_Handle is a system wide unique number that is assigned to each segment with an entry in the G_AST (viz., every active segment). This "handle" is the instrument of controlled single copy sharing of information (segments). It allows a segment to exist under one unique handle but be accessible in the address space of more than one process (with different segment numbers in each address space). The MM_Handle is returned to the Segment Manager by the Memory Manager during: the execution of the Make Known instruction.

The Size field is an integer value (of language structure type "word") which represents the number of 256 byte blocks composing a segment.

The Access_Mode field is used to describe the process' access to the segment (i.e., null or read and/or write).

The In_Core field is used to indicate if the segment is or is not in main memory (i.e., this field is a flag or true/false boolean switch).

The Class field is a long word field used to represent the degree of information sensitivity (viz., access class) assigned to the segment. This field (for example) would be used to numerically describe a classification label (as described above).

```
KST_REC Record [MM_Handle Array [3 Word]
Size Word
Access_Mode Byte
In_Core Byte
Class Long
M_Seg_No Short_Integer
Entry_Number Short_Integer]
```

Figure 41: Known Segment Table

The Mentor_Seg_Nr field is a number representing the segment number of a segment's parent or "mentor" segment.

Its importance will discussed shortly.

The Entry_Nr field is a number representing a segment's index number into its parent or mentor segment's Alias Table (not yet discussed).

The Alias Table is a Memory Manager database and will be described later. The aliasing scheme provided via the aliass tables is used to prevent passing system wide information out of the Kernel (i.e., the Unique_ID of a segment). The "alias" of a segment is the concatenation of the Mentor_Seg_Nr with the segment's Entry_Nr (index) into the mentor segment's Alias Table. It is clear that the last two fields of a KST record are the "alias" of that segment.

B. NON-DISCRETIONARY SECURITY MODULE

The key in protection of secure information using internal controls was identified as the security kernel concept. The basic idea within this concept is to prove the hardware part of the Kernel correct and, similarly, to keep the software part small enough so that proving it correct is feasible. A central component of the kernel software is the Non-Discretionary Security Module (hereafter referred to as the NDS Module). The NDS Module is concerned only with the non-discretionary aspect of the security policy in effect; since the discretionary aspect is subservient in nature to

the non-discretionary aspect, it is then sufficient that the Kernel contain only the software representing the non-discretionary aspect of the security policy. The discretionary security is provided outside the kernel in the SASS supervisor. Every attempt to access information must result in an invocation of the NDS Module.

The function of the NDS Module is to compare two classifications (viz., compare two labels), make a decision as to their relationship (i.e., =,>,<,i), and return a true/false interpretive answer relative to the query of the calling procedure. The mechanism used as a basis is the lattice model abstraction previously discussed. The NDS module does not require a database since the labels it compares are stored in (passed from) other Kernel databases.

C. MEMORY MANAGER

1. Function

The Memory Manager process is the only component of the non-distributed kernel. It is responsible for managing the real memory resources of the system -- main (local and global) memory and secondary storage. It is tasked by other processes within the Kernel domain (via Signal and Wait) to perform memory management functions. This thesis will address the Memory Manager in terms of two components: (1) the Memory Manager Process (also called the nondistributed kernel and the Memory Manager Module), and (2) the distributed

Memory Manager (also called the Distributed Memory Manager Module). The former is the "true" memory manager while the latter is the interface with other processes, that is, it resolves the issue of interprocess communication with the "true" memory manager.

The Distributed Memory Manager Module creates the following extended instruction set: MM CREATE ENTRY, MM DELETE ENTRY, MM ACTIVATE, MM DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The instructions form the mechanism of communication between the Segment Manager of a process and a memory manager process (where the actual memory management functions are performed). The Memory Manager Process instruction set corresponds one to one with that of the Distributed Memory Manager; the set consists of: CREATE ENTRY, DELETE ENTRY, ACTIVATE, DEACTIVATE, SWAP IN, and SWAP OUT. The basic functions performed by the Memory manager are allocation/deallocation of global and local memory and of secondary storage, and segment transfers from local to global memory (and vice-versa) and from secondary storage to main memory (and vice-versa).

2. Databases

A detailed and descriptive discussion of the Memory Manager databases is presented in the work of Gary and Moore [5], and the reader may refer to it for memory manager database details. This thesis addresses the implementation of

the distributed Memory Manager but not the Memory Manager Process, thus brief descriptions are provided of the latter's databases.

The Global Active Segment Table (G_AST) is a system wide (i.e., shared by all memory manager processes) database used to manage all active segments. A lock/unlock mechanism is used to prevent race conditions from occurring. The distributed memory manager of the signalling process locks the G_AST before it signals the memory manager process.

The Local Active Segment Table (L_AST) is a processor local database which contains an entry for each segment active in a process currently loaded in local memory.

The Alias Table is a system wide database associated with each nonleaf segment in the Kernel. It is a product of the aliasing scheme used to prevent passing system wide information out of the Kernel. The alias table header (provided for file system reconstruction after system crashes) has two pointers, one linking the alias table to its associated segment, the other linking the alias table to the mentor segment's alias table. The fields in the alias table are Unique_ID, Size, Class, Page_Table_Loc, and Alias_Table_Loc. The index into the alias table is Entry_No.

The Memory Management Unit Image (MMU_Image, Figure 42) is a processor local database indexed by DBR_No (viz., for each DBR_No there is a MMU_Image record, with each record containing a software image of the segment descriptor regis-

ters of the hardware MMU). The MMU_Image is an exact image of the MMU. Each record is indexed by Segment_No (segment number) and each Segment_No entry contains three fields. The Base_Addr field contains the segment's base address in memory. The Limit field contains the number of blocks of contiguous storage for the segment (zero indicates one block). The Attributes field contains 8 flags including 5 which relate to the memory manager. The Blks_Used field and the Max_Blks (available) fields are per record (not per segment entry) and are used in the management of each process' virtual core.

The Memory Bit Maps (Disk_Bit_Map, Glo-bal_Memory_Bit_Map, and Local_Memory_Bit_Map) are memory block usage maps that use true/false flags (bits) to indicate the use or availability of storage blocks.

The only database in the Distributed Memory Manager is; the Memory Manager CPU Table (Figure 43). It is an array of memory manager VP_ID's (MM_VP_ID) indexed by CPU number. This table enables a signalling process to identify the appropriate memory manager process (virtual processor) to signal.

	Blocks Used	
	Max Avail Blocks	
Segment	Base Addr Limit	Attributes
1	1	1 1
v,		
	1	
		1

```
MMU_Image Array [Max_DBR_No MMU]

Record [SDR Array [No_Seg_Desc_Reg Seg_Desc_Reg]]

Blks_Used Word Max_Blks Word]

Seg_Desc_Reg Record [Base_Addr Address Limit Byte Attributes Byte]

Address Word
```

Figure 42: Memory Management Unit Image

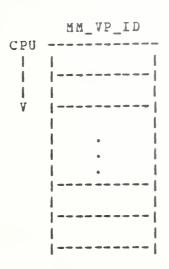


Figure 43: Memory Manager-CPU Table

D. SUMMARY

The segment management functions and key related concepts (such as segmentation) were discussed in this chapter.

The importance of segmentation to data sharing and information security was emphasized as were key information security concepts. With this background, the implementation of segment management and a non-discretionary security policy will be described in the following chapter.

Chapter WVIII

SEGMENT MANAGEMENT IMPLEMENTATION

The implementation of segment management functions and a non-discretionary security policy is presented in this chapter. Paramount to this implementation were several key issues that affected the implementation. These issues are discussed first. The implementation is discussed in terms of the Segment Manager, Non-Discretionary Security (NDS), and Distributed Memory Manager modules.

A. IMPLEMENTATION ISSUES

Segment management for the SASS was provided through the implementation of the Segment Manager Module, the NDS Module, and the Distributed Memory Manager Module. Additionally, since a demonstration/testbed was integral to the testing and verification of the implementation, it was necessary to complete other supportive tasks. Reitz [12] provided a demonstration of the operation of the Inner Traffic Controller primitives SIGNAL and WAIT (for interprocess communication). Integral to this demonstration was the correct performance of the Inner Traffic Controller VP scheduling mechanism and a "stub" of the Traffic Controller and its process scheduling mechanism (the TC support and use of the

mechanism of eventcounts and sequencers was not a part of the demonstration). The Segment management demonstration (hereafter referred to as "Seg_Mgr.Demo") was "built on top of" Reitz' ITC synchronization primitive demonstration (hereafter referred to as "Sync. Demo"). Thus, an immediate issue was to resolve the feasibility of adding on to Sync.Demo and also to refine the present design of the Sync. Demo to facilitate its integration into the Seg_Mgr.Demo. One aspect of this effort was in resolving the problem of how to pass (i.e., in interprocess communication) a larger message.

1. <u>Interprocess Messages</u>

The Sync.Demo passed "word" (16 bit) messages. To provide the mechanism for the distributed memory manager to signal the memory manager process with a command function identification code and the arguments needed to perform that function (e.g., CREATE-ENTRY and its input arguments), a message size of at least eight words (16 bytes) was necessary. An obvious answer was to signal with an array of eight words as the message. PLZ/SYS, however, does not allow passing arrays in its procedure calls (a procedure call is analogous to a subroutine call). Another alternative was to signal with a pointer to the array of words, since PLZ/SYS does allow passing pointers in procedure calls (thus the message would be a pointer to the real message). This,

however, would be invalid in the segmented implementation (on the Z8000 segmented microprocessor) since identical segment numbers in different processes may not refer to identical segments. For example, a pointer in a process (e.g., file management) points to an array (i.e., provides its address) by segment number and offset; passing this pointer to another process (e.g., memory manager) would provide this same segment number and offset which, of course, may be a different object in the second process's address space).

Another alternative considered was that of a shared "Mailbox" segment with an associated eventcount acted on by the Kernel Inner Traffic Controller primitives TICKET, ADVANCE and AWAIT. A design for using this concept in the supervisor ring is provided by Parks [9]. This alternative was not deeply considered since these primitives are not included in the current Inner Fraffic Controller.

The method ultimately used to signal the new length messages is based on the fact that the ITC is in both the signalling and the receiving (memory manager) processes address space. The message is loaded into an array in process #1 and a pointer to the array is passed in the call SIGNAL; the VPT, the ITC's database, is then updated by (using the pointer) putting the message into its MSG_Q section. The message is retrieved by process #2 by execution of Reitz' WAIT primitive with only one refinement. That refinement is for the "waiting" process to provide as an argument (in the

WAIT primitive) a pointer to its own message array so that the message in the VPT can be copied to it. This refinement provides for passing a long message essentially "by value" between processes.

2. Structures as Arguments

Another issue concerned the use of pointers in the implementation of segment management. This necessary "evil" is a result of the need to pass linguistically "complex" data types in procedure calls. Complex types refer to array and record structures in PLZ/SYS (as opposed to the "simple" types--byte, word, integer, short-integer, long, and pointer). In managing databases (e.g., KST, G_AST) which consist of arrays of records (which in turn contain records and/or arrays), it was frequently necessary to reference data as an array or record. Within a process, the use of pointers was not a problem (i.e., not a problem such as would be encountered in IPC passing of pointers).

3. Reentrant Code

The issue of code reentrancy was addressed at the assembly language level through the use of a stack segment and registers for storage of local variables. PLZ/SYS (high level language) does not address reentrant procedures and thus the segment management high level code is not automatically reentrant. The problem of reentrancy can be seen by

looking at a shared procedure that is not reentrant; such a procedure has storage for its variables allocated statically in memory. Suppose a procedure (e.g., in the Kernel) can be activated by more than one process. While the procedure is executing in one process, a process switch occurs (e.g., to wait for a disk transfer) and its execution is suspended. The second process is activated, and while it is running it invokes the procedure. While the procedure is executing for the second process it uses the same storage space for variables as it did when executing for the first process. Eventually, it relinquishes the processor. However, when the procedure resumes its execution for the first process, the variable values that were in use by it originally have been changed during its execution in the second process. Thus, incorrect results are now inevitable.

4. Process Structure of the Memory Manager

References to the "Memory Manager" in past works have generally meant the memory manager process (non-distributed kernel). This work references two distinct components of the "memory manager module". The Distributed Memory Manager is an interface provided to the Memory Manager Process. It is, in fact, distributed in the address space of each Supervisor process. In contrast, the Memory Manager Process clearly is not distributed and its address space is contained entirely in the Kernel.

5. Per-Process Known Segment Table

Another key issue was that of the per process Segment Manager database, the KST. Since each process has its own KST, it cannot be linked to the (shared) segment manager procedures. To implement the KST as a per process database, it was convenient to establish, by convention, a KST segment number that is consistent from process to process. That segment in each process is the KST segment for that process. Implementation is then accomplished by using the segment number to construct a pointer to the base of the appropriate KST. It is then easy to calculate an appropriate offset to index any desired entry in the KST data.

6. DBR Handle

In Reitz's implementation of the multilevel scheduler and the IPC primitives, references to "DBR" (descriptor base register) are references to an address. That address value represents a pointer to an MMU_IMAGE record containing the list of descriptors for segments in the process address space. Gary and Moore [5] reference a "DBR_NO" that is essentially a handle used within the memory manager as an index within the MMU_IMAGE to a particular MMU record. The base address of the MMU record indexed by DBR_NO is then equivalent to the concept of DBR value used in Reitz' work. The effect of this inconsistency on the segment management implementation was minor and will be further discussed later in this chapter.

B. SEGMENT MANAGER MODULE

The Segment Manager Module consists of six procedures representing the six extended instructions it provides. These are based on the design of Coleman [2]. Only calls from external to the Kernel (via the Gate Keeper) made to the Segment Manager (per the loop-free structure of the SASS). The normal sequence of invocation of the Segment Manager functions to allow referencing a segment is: (1) CREATE_SEGMENT--allocate secondary storage for the segment update the mentor segment's Alias Table, MAKE_KNOWN--add the segment to the process address space (segment number is assigned), (3) SWAP IN--move the segment from secondary storage into the process's main memory. normal sequence of invocation to "undo" the above is: SWAP_OUT--move the segment from main memory to secondary storage, (2) TERMINATE--remove the segment from the process's address space, (3) DELETE_SEGMENT--deallocate secondary storage and remove the appropriate entry from the alias table of its mentor segment. The six Supervisor entries into the Segment Manager (viz., the six extended instructions) will be discussed individually below. The PLZ/ASM listings for the Segment Manager are in Appendix H.

1. Create a Segment

The function that creates a segment (i.e., adds a new segment to the SASS) is CREATE_SEGMENT. This function validates the correctness of the Supervisor call by checking the parameters and making certain security checks. The distributed memory manager is then called to accomplish interprocess communication with the Memory Manager Process, where segment creation is realized through secondary storage allocation and alias table updating.

passed as arguments: (1) Men-CREATE SEGMENT is tor_Seq_No--the segment number of the mentor segment of the segment to be created, (2) Entry No--the desired entry number in the alias table of the mentor segment, (3) Class--the access class (label) of the segment to be created, and (4) Size--the desired size of the segment (in blocks of initial check is to verify that the desired: The size does not exceed the designed maximum segment size. this check is satisfactory, a conversion of the Mentor_Seq_No to a KST index is necessary. This is because the Kernel segments use the first several segment numbers available but do not have entries in the KST. Thus if there were 10 Kernel segments and a system segment had segment number 15, then its index in the KST would actually be 5 (i.e., the Kernel segments would use numbers 0-9, and this segment would be the sixth segment in the KST and its index would be 5). A call is then made to the procedure

ITC_GET_SEG_PTR with the constant KST_SEG_NO passed as parameter. This procedure will return a pointer to the base of this process' KST. This pointer is then the basis for addressing entries in the KST. The next check is to see if the mentor segment is known (viz., is in the address space of the process, and thus, in the KST). The key to determining if any segment is known is the mentor segment entry (M_SEG_No) for that segment in the KST. If not known, this entry in the segment's KST record will be filled with the constant NULL_SEG. The basis for checking to see if the segment's mentor segment is known is the aliasing scheme implication that a mentor segment must be known before a segment can be created. The process classification must next be obtained from the Traffic Controller. The process classification is checked to ensure that it is equal to the classification of the mentor segment since write access to its alias table is needed to create a segment. The NDS module's CLASS_EQ procedure is called and returns a code of true or false. The last check is the compatibility check to ensure that the classification of the segment to be created is greater than or equal to the classification of the mentor segment. This is accomplished by calling the NDS Module's CLASS_GE procedure which returns a code of true or false. If any of these checks are unsatisfactory, an appropriate error code is generated and the Segment Manager returns to its calling point. If all checks are satisfactory, then a

pointer to the mentor segment's MM_Handle array is derived (HPTR). Note that in the current memory manager design [5] the actual MM_Handle contents are a Unique_ID (a long word, viz., two words concatenated), and an Index_No (index into the G_AST, a word); thus together these two fields are a total of three words. Since the Segment Manager does not interpret this handle, it is considered a three word array at this level. For this reason, the entire uninterpreted MM_Handle array will be passed by passing its pointer. pointer and Entry_No, Size, and Class are then passed in a the distributed memory manager procedure call to MM_CREATE_ENTRY. This procedure, in turn, performs IPC with the memory manager process where segment creation ultimately is accomplished. A success code is returned in an IPC message from the memory manager process via the distributed memory manager to the CREATE_SEGMENT procedure to indicate success or failure as appropriate. This success code is checked by the Segment Manager to ensure confinement would not be violated if it is returned to the calling process! supervisor domain. Only after the success code has been returned can the action of segment creation be considered com-Segment creation does not imply the ability to reference that segment; MAKE_KNOWN will accomplish that.

2. Delete a Sequent

The function that deletes a segment (i.e., deletes a segment from SASS) is DELETE_SEGMENT. Validation of parameters and security checks are performed here similar to (but fewer than) the CREATE_SEGMENT checks. The distributed memory manager is then called to cause IPC with the memory manager process, where segment deletion is realized through secondary storage deallocation and alias table entry deletions. DELETE_SEGMENT is passed as arguments: (1) tor_Seg_No and (2) Entry_No. Conversion of the Mentor_Seg_No to a KST index is accomplished first. The pointer to the base of the KST is located and returned. as before. The mentor segment is checked to ensure it known, again, by verifying that its own M_SEG_No (mentor segment number) entry in the KST is not the NULL_SEG. The process classification is obtained from the TC and checked (by a call to CLASS_EQ) to ensure it is equal to the mentor segment classification, since deleting an entry requires write access to the alias table. If all checks are satisfactory, then the mentor segment's MM_Handle pointer is derived. This pointer and the mentor segment alias table entry number are passed in a call to the distributed memory manage er procedure MM DELETE ENTRY. It then performs IPC with the memory manager process where segment deletion is accomplished and a success code is returned as before.

3. Make a Segment Known

The function that makes a segment known (i.e., adds that segment to the process' address space by assigning a segment number, updating the KST, and causing the memory manager process to "activate" the segment (that is, add it to the AST)) is MAKE_KNOWN. Making a segment known is the way the supervisor declares its intention to use a segment.

MAKE_KNOWN is passed as arguments: (1) Mentor_Seg_No, (2) Entry_No, and (3) Acess_Desired (e.g., write, read, or null). It returns (1) a success code, (2) the access allowed to the segment, and (3) the segment number. Conversion of the mentor segment number to a KST index, finding the KST pointer, and verifying that the mentor segment is known occur as previously discussed.

There are three basic cases that may occur in MAKE_KNOWN: (1) the segment is already known (has an entry in the KST), (2) the segment is not known and there is a segment number available, or (3) the segment is not known and there is no segment number available.

A search is made of the KST using each record's (segment's) M_SEG_No (mentor segment number) and Entry_Number fields as the search key. If these two fields match the input values Mentor_Seg_No and Entry_No, then the record indexed is that of the desired segment; thus the segment to be made known is already known. In this case, all that need be done is to return the success code, segment number (convert-

ed from the index by adding to it the number of kernel segments), and the access allowed (equal to the Access_Mode entry in the KST for the already known segment).

During the search of the KST, the M_SEG_No field is also checked to see if it contains the NULL_SEG entry (this implies that the segment number associated with the record is "available"). The first time this is noted, the index is saved. Note the first available index is saved since it is desired to assign segment numbers at the "top" of the KST to keep it dense there. When the search does not find that the segment is already known, the index for the available segment number is retrieved and converted to segment number by adding to it the number of kernel segments. If this index is the NULL_SEG entry, then there is no segment number available. In this event, the success code is set to NO_SEG_AVAIL, the segment number is assigned NULL_SEG, and access allowed is set to NULL_ACCESS (this is the third case mentioned). If the index is not equal to NULL_SEG and conversion to segment number has occurred then the Traffic Controller is called to provide the DBR_No (descriptor base register number) for the current process. The DBR_No is used by the memory manager process as an index in the MMU_Image and the local AST. The distributed memory manager procedure MM Activate is called: it is passed the DBR number, the pointer to the mentor segment's MM_Handle entry, the mentor segment alias table Entry_No, and the segment

number. MM_Activate performs the normal interface function (performs IPC with the memory manager process procedure that updates the local and global AST's) and also updates the KST entry for the new segment's MM_Handle entry (returned from the memory manager process). It also returns to the Segment Manager the success code, the segment classification, and the segment size from the memory manager process. If the success code is "succeeded" then the issue of access to be granted must be resolved. The process classification is obtained from the TC and passed with the segment classification to the NDS Module procedure CLASS_GE. Ιf CONDITION_CODE returned is FALSE then access allowed NULL ACCESS. the segment number is NULL SEG, and MM_DEACTIVATE is called to deactivate the segment. An appropriate error code is returned. If it is greater than or equal then the access allowed is assigned as follows: the two classifications are compared again--this time to see if equal; (2) If they are equal, then the access allowed is either read or write per the access desired: (3) if they are not equal (i.e., the process class is greater than the segment class) then the access allowed is read. Finally the KST entries for that segment number (more accurately for its index in the KST) are filled with the appropriate information (e.g., IN_CORE is false, etc.). If the success code returned from the memory manager process via the distributed memory manager is not "succeeded", then the segment number

is set to NULL_SEG and the access allowed is set to NULL_ACCESS.

4. Make a Segment Unknown (Terminate)

The function that makes a segment unknown (i.e., removes that segment from the process' address space--by updating the KST and causing the memory manager process to "deactivate" the segment) is TERMIMATE. It results in removal of the M_SEG_No (mentor segment number) entry from that segment's KST record. Terminate is passed the segment number of the segment to be terminated as an argument. It returns a success code. Conversion of the segment number to a KST index, finding the KST pointer, and verifying that the segment is known occurs in the same manner as previously discussed. The next check is to verify that the segment is not still lcaded in the process' virtual core (viz., it has been "swapped-out"). If not, an error code is returned and the user must cause the Segment Manager extended instruction SM_SWAP_OUT to be executed. The next check is to ensure that the user is not attempting to terminate a Kernel segment. The first several segment numbers in a process' address space will be used by Kernel procedures and data (though they will not be entries in the KST). Thus if there vere 10 Kernel segments, then the segment number to be terminated must be greater than or equal to #10 (since the Kernel segments used #'s 0-9). Thus a check is made to ensure

that the segment number is not less than the number of Kernel segments; otherwise an error code is returned. Next, the segment number is checked to ensure that it is not larger than the maximum segment number allowable (if so, an error code is returned). If all checks are satisfactory, then the segment's MM_Handle pointer and the process DBR_No are obtained (as discussed before) and passed in a call to the MM_Deactivate procedure. It calls the memory manager process procedure DEACTIVATE which removes or updates (as appropriate) the entries in the local and global AST's.

5. Swap a Sequent In

The function that swaps a segment from secondary storage to main memory (global or local) is SM_SWAP_IN. It is passed the segment number of the segment to be swapped in as an argument and returns a success code. Conversion of the segment number to a KST index, finding the KST pointer, and verifying that the segment number is known are accomplished as previously discussed. If the check is satisfactory, then the segment's MM_Handle pointer and the process DBR number are obtained. They are passed with the segment's access mode (from the KST) as arguments in the call to MM_SWAP_IN. It performs normal interface (IPC) functions and returns a success code from the memory manager process' SWAP_IN procedure (where, if not already in core, allocation of main memory space and reading the segment into main memory occurs).

If the success code is "succeeded" then the segment's IN_CORE entry in the KST is updated to show that the segment is in main memory for this process (i.e., the entry is now "true").

6. Swap a Segment Out

The function that swaps a segment from main memory to secondary storage is SM_SWAP_OUT. It is passed the segment number of the segment to be swapped out as an argument and returns a success code. The behavior of SM_SWAP_OUT is exactly analogous to that of SM_SWAP_IN except that the segment's KST IN_CORE entry is updated to reflect that the segment has been removed from main memory for this process (i.e., the new entry is "false").

C. NON-DISCRETIONARY SECURITY MODULE

The Non-Discretionary Security Module implements the non-discretionary security policy for the SASS. The NDS module contains two procedures: CLASS_EQ and CLASS_GE; both compare two labels (classifications) and determine if their relationship meets that of the procedure's name (i.e., equal, or greater than or equal). Although the type of checks being made are, in fact, compatibility checks, Simple Security Condition checks, etc, the NDS Module does not recognize or need to recognize this. It simply uses an algorithm to determine if classification #1 = classification #2

or if classification #1 >= classification #2, as appropriate. It then returns a condition code of true or false in accordance with the particular case. The earlier discussion of label comparison in accordance with a partially ordered lattice structure is relevant in discussing the NDS Module's algorithm. Consider the same "totally ordered" relationship TS > S > C > U of levels and the "disjoint" relationship Cy | N | Nu | % of categories. Comparison of levels will be numerical comparisons while comparison of categories will use set theory comparison as a basis. If TS=4, S=3, C=2, U=1 are level numerical assignments, then the totally ordered relationship is maintained (i.e., TS>S>C>U is still true). Now consider the categories and make the following assignments: Cy=1, N=2, Nu=4, %=0. Note that a classification may have only one level and one category set (the category set may contain several categories). Consider this example: (TS, Cy,N. The level is TS (=4). The category is the set Cy,N and numerically is formed by performing a logical OR with the categories Cy and N. Sixteen bit representation of this is:

Cy OR N

(0000 0000 0000 0001) OR (0000 0000 0000 G010)

- $= 0000 0000 0000 0011 = Cy_N$
- If (TS, Cy,N) is considered label #1 and (S, N) as label #2 then a comparison of the two labels would be:
- (1) Compare level #1 with level #2 -- 4 > 3?
 Clearly, the answer is yes.

(2) Compare category #1 with category #2 -- is
 (0000 0000 0000 0011) a superset of
 (0000 0000 0000 0010), or more clearly
 is the latter a subset of the former?

The answer is yes, and one way to show that is true is by performing a logical OR of category #1 with category #2 and comparing the result to category #1. If the result of the OR operation equals category #1 then category #1 is a superset (not necessarily proper) of category #2. Since usage of the term subset is more frequent than that of superset, this relationship will typically be stated as "category #2 is a subset of category #1. To illustrate the above:

(0000 0000 0000 0011) OR (0000 0000 0000 0010)

= 0000 0000 0000 0011 = category #1.

Cy, N OR

This means, in this example, that category #2 is a subset (not necessarily proper) of category #1. Since level #1 > level #2 and category #2 subset category #1 then label #1 > label #2. Thus, a call to the CLASS_EQ procedure with these two labels as the input classifications would return a condition code of false while CLASS_GE would return true. The decision to have the classifications as long word (32 bits) supports the requirement of some DoD specifications for eight levels and sixteen categories. This module uses sixteen bits for the level and sixteen bits for the category. Appendix I is the PLZ/ASM listings for the NDS Module.

1. Equal Classification Check

The CLASS_EQ procedure performs comparison of two classifications (labels) and returns a condition code of true if they are equal (an exact match of the two long words bit per bit) or false if they are not.

2. Greater or Equal Classification Check

The CLASS_GE procedure performs comparison of two classifications (labels) and returns a condition code true if classification #1 is greater than or equal to classification #2 or a condition code of false otherwise. For classification #1 to be greater than or equal to classification #2, the following must be true: (1) level #1 >= level #2 (determine this by simple numerical comparison of values) and (2) category #2 subset category #1 (determine this by performing a logical OR with the categories and comparing the result to category #1 -- if they are equal then category #2 is a subset of category #1).

Since PLZ/SYS allows passing only "simple" types in calls, the labels were passed as long words (as opposed to each being word arrays of length two). An access class label is never interpreted outside the NDS Module. However, within the NDS Module it is necessary to address the classification's components separately (viz., level and category). Thus, an "overlay" of the logical view of the classification was created. This overlay was a record of type ACCESS_CLASS

and it consisted of two fields: level -- 16 bit integer and category -- 16 bit integer. A pointer type CPTR was declared to be of type pointer to ACCESS_CLASS. Two other pointers CLASS1_PTR and CLASS2_PTR were declared to be of type CPTR and were set equal to the base address of CLASS1 and CLASS2 respectively. This "overlay" of the record frame over the two classification labels passed as arguments allowed the desired component addressibility. Puthermore, the non-discretionary policy enforced by SASS can be changed from the current DoD policy to another lattice policy by changing (only) the NDS Module.

D. DISTRIBUTED MEMORY MANAGER MODULE

The Distributed Memory Manager Module performs as an interface between the Segment Manager and the Memory Manager Process. As its name implies, it is distributed in the kernel domain of each Supervisor process. The key role performed in this module is to arrange and perform interprocess communication between its process (actually the VP) and the memory manager process (VF). The module consists of eight procedures. Six of the procedures are called directly by Segment Manager procedures; they are MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The other two procedures are "service" procedures called by multiple procedures; they are:

first six procedures is somewhat uniform (except for MM_ACTIVATE). Thus, the general logic will be explained (with MM_CREATE_ENTRY as an example) and it should suffice as a description for all (except MM_ACTIVATE) procedures. The service procedures will be described separately.

1. Description of Procedures

Each procedure is invoked (and returns) on a one to one basis with a corresponding procedure in the Segment Manager. For example, CREATE_SEGMENT invokes MM_CREATE_ENTRY which signals the CREATE_ENTRY procedure in the Memory Manager Process Module. Associated with each procedure is an IPC message "frame" to describe the unique format of the contents of the message to be signalled to the memory manager Similarly, there must be a message "frame" for return messages from the memory manager process; this frame is the same for all but the MM_ACTIVATE procedure. Consider the message frame for MM_CREATE_ENTRY; it consists of: (1) a code to describe which function is to be performed (e.g., CREATE_CODE indicates that the CREATE_ENTRY procedure is the intended recipient of the message), (2) MM_Handle (an array of three words), (3) Entry_No, (4) Size, and (5) Class. The message frame has a filler (in this case) of one byte to ensure that it is of length 16 bytes. The purpose of this frame is to provide an overlay onto the actual message array to be signalled and to facilitate loading the arguments into

the message array. This is accomplished by having a pointer of the type that points to the frame but by converting its address so that it actually points to the base of the message array. Consider these lines of PLZ/SYS code:

CE_MSGPTR := CE_PTR COM_MSGPTR

CE_MSGPTR-. CREATE_CODE := CREATE_ENTRY_CODE

This code is putting a value into the structure pointed to by CE_MSGPTR at entry CREATE_CODE. The key point is that the frame of that structure is, in fact, CREATE_MSG (as described before), but the physical location pointed to is the message array. This is assured by having the pointer CE_MSGPTR (which points to a structure of type CREATE_MSG) set equal to a pointer (COM_MSGPTR) to the actual message array (COM_MSGBUF). This is accomplished by the first line of code. The message array itself is never directly referenced, but rather the message array that is overlayed by the message frame is filled in the format of the CREATE MSG frame. In this example, the first two bytes of the message array now contain the value of the constant CREATE_ENTRY_CODE. The remainder of the message array is filled in the same manner (all procedures use the same notion of a frame, although the frames have different formats). The PERFORM_IPC (perform interprocess communication) procedure is called by all procedures at this point in their execution. The key is that the argument passed is the message array pointer not the pointer to the CREATE_MSG record

(after all it is only an overlay frame -- linguistically, it is only a type and is never declared as a structure requiring memory storage allocation). When PERFORM_IPC returns, the message array contains a return message. This message consists of only a success code and filler space in all cases but MM_ACTIVATE. Interpretation of the return message is performed in the same manner as loading the message array. The retrieved success code is returned to the calling Segment Manager procedure. For MM_ACTIVATE, the return message must be interpreted and values for success code, segment size, and segment classification retrieved and returned to the Segment Manager MAKE_KNOWN procedure. The value for the MM_Handle (called the G_AST_Handle by the memory manager process) must be retrieved and entered in the KST record for this segment.

2. Interprocess Communication

The final arrangements and actual performance of IPC is completed by the internal procedure PERFORM_IPC. By locating the identity of the current physical processor (CPU) and using that identity to index into the MM_CPU_TABLE, the VP_ID of the current memory manager is resolved, so that the memory manager process dedicated to this physical processor is signalled. The call to K_LOCK is, in fact, a disguised call to the SPIN_LOCK procedure (since K_LOCK calls SPIN_LOCK). K_LOCK represents an ultimate (as yet unimplemented) goal of

a Kernel locking (wait-lock) system. In any event, the G_AST lock must be set prior to signalling the memory manager process. After SIGNAL has been called, a call is made to WAIT with the pointer to the message array as the argument. The synchronization cycle that results is: (1) PERFORM_IPC calls the ITC procedure SIGNAL with the memory manager VP ID and message array pointer as arguments; PERFORM_IPC then calls WAIT with the message array as the argument, SIGNAL causes the message array to be copied into the message queue (in the VPT) of the appropriate VP ID, (3) ultimately, the signalled VP is scheduled; it had previously called WAIT, passing a pointer to its own local message array; the action of WAIT is to copy the message from the VPT to the signalled process' local message array: there it is interpreted by the memory manager process main procedure and the appropriate procedure is called for action (e.g., CREATE_ENTRY), (4) when action is completed the memory manager process fills its local message array with the appropriate return message and calls SIGNAL with a pointer to the message and the original signalling process' VP_ID as arguments. (5) SIGNAL causes the memory manager process' message to be copied into the VPT message queue for the appropriate VP ID. (6) that VP is eventually scheduled and through the action of WAIT has the return message copied from its message queue in the VPT to its local message array; WAIT then returns to PERFORM_IPC. The G_AST lock is unlocked and PERFORM_IPC returns to the appropriate distributed memory manager procedure.

The last procedure in the distributed memory manager is MM_GET_DBR_VALUE. This procedure simply provides the service of translating a DBR_NO (DBR number) into its appropriate DBR address. It is called by the TC_GETWORK procedure to allow it to call the ITC procedure SWAP_VDBR (remember that presently the Inner Traffic Controller deals with the DBR as the address of the appropriate MMU record in the MMU_IMAGE while the Traffic Controller uses DBR as a DBR number which indexes to the appropriate MMU record).

E. SUMMARY

The implementation of segment management functions and a non-discretionary security policy for the SASS has been presented in this chapter. The implementation of the Segment Manager Module, Non-Discretionary Security Module, and Distributed Memory Manager management demonstration was described.

Chapter XIX

CONCLUSIONS AND FOLLOW ON WORK

The implementation of segment management for the security kernel of a secure archival storage system has been presented. The implementation was completed on Zilog's Z8002 sixteen bit nonsegmented microprocessor. Segmentation hardware (Zilog's Z8010 Memory Management Unit) was not available, therefore it was simulated in software as described by Reitz [12]. The loop free modular construction used in the implementation facilitates ease of expansion or modification.

A non-discretionary security policy was implemented using a partially ordered lattice structure as a basis. Enforcement was realized through an algorithm that compared two labels and determined if their relationship was equal to a desired relationship. Although the DoD security classification system was represented, any non-discretionary security policy that may be represented by a lattice structure may similarly be implemented. This implementation has shown that by having the non-discretionary security policy enforced in one module, changing to another policy requires changing only this one module.

Software engineering techniques used in previous work emphasized the advantages of working with code that is well structured, well documented, and well organized. Despite being written in assembly language, Reitz implementation of multiprogramming and process management proved to be consistent in style, clarity and documentation. This enhanced the construction of a segment management demonstration which was built onto his synchronization demonstration. Further, refinements made to his code (not necessitated by any failures of his code) were relatively easily accomplished.

while the segment management implementation appears to perform properly, it has not been subjected to a formal test plan. Such a test plan should be developed and implemented.

The Memory Manager Process has been designed but not implemented. Segment management implementation, provision for IPC using more practical size messages, and the detailed design of the memory manager by Moore and Gary [5], provide a sound foundation for memory manager implementation. A framework of the mainline code needed is provided in the Memory Manager Module of the demonstration code in Appendix J. Prior to this implementation, formal testing of the segment management implementation herein and the monitor implemented by Reitz [12] should be completed.

PART F

IMPLEMENTATION OF PROCESS MANAGEMENT FOR A SECURE ARCHIVAL STORAGE SYSTEM

This section contains excerpts from a Naval Postgraduate School MS Thesis by A. R. Strickler [19]. The origins of these excerpts are:

INTRODUCTION from Chapter I
IMPLEMENTATION ISSUES from Chapter III
PROCESS MANAGEMENT IMPLEMENTATION from Chapter IV
CONCLUSION from Chapter V

Minor changes have been made for integration into this report.

Chapter XX

INTRODUCTION

This thesis addresses the implementation of process management functions for the Secure Archival Storage System or SASS. This system is designed to provide multilevel secure access to information stored for a network of possibly dissimilar host computer systems and the controlled sharing of data amongst authorized users of the SASS. Effective process management is essential to insure efficient use and control of the system.

The major accomplishments of this thesis effort include the provisions for efficient process creation and management. These functions are provided through the establishment of a system Traffic Controller and the creation of a virtual interrupt structure. An effective mechanism for inter-process communication and synchronization is realized through an Event Manager that makes use of uniquely identified segments supported by eventcount and sequencer primitives. A hardware controlled two domain operational environment is created with the necessary interfacing between domains provided by a software "gate" mechanism. Additional support is provided through considerable work in the area of database initialization and a technique for limited dynamic memory allocation.

This implementation was completed on the commercial AMC Am96/4116 MonoBoard Computer with a standard Multibus interface.

Chapter XXI

IMPLEMENTATION ISSUES

Issues bearing on the implementation of process management and refinements made to existing modules are presented in this chapter. Process management for the SASS was provided through the implementation of the Traffic Controller Module, the Event Manager Module, the Distributed Memory Manager Module, and a Gate Keeper Stub (system trap). Additionally, since a demonstration/testbed was integral to the testing and verification of the implementation, it was necessary to complete other supportive tasks. These supportive tasks included limited Kernel database initialization, revised preempt interrupt handling mechanisms, Idle process definition and structure, and additional refinements to existing modules.

A. DATABASE INITIALIZATION

Previous work on SASS has relied on statically built databases, which proved to be sufficient for demonstration of a single processor, single host supported system. In the current demonstration, multiple hosts are simulated, and the Kernel data structures have been refined to represent a multiprocessor environment. Since a multiprocessor system was

unavailable at the time of this demonstration, several "runs" were made and traced, using different logical CPU numbers, to show the correctness of this structure. Due to this multiprocessor representation and simulation of multiple hosts, the use of statically built Kernel databases was no longer convenient. Therefore, it became necessary to provide initialization routines for the dynamic creation of those Kernel databases required for this implementation. While it was not the intent of this effort to implement system initialization, care was taken in the writing of these initializing routines so that they might be utilized in the system intialization implementation with, hopefully, minimal refinement. Database initialization was restricted to those databases existing in the Inner Traffic Controller and the Traffic Controller. Limited elements of the Known Segment Table (KST) and Global Active Segment Table (G_AST) were also created for demonstration purposes.

1. Inner Traffic Controller Initialization

A "Bootstrap Loader" Module, which logically exists at a higher level of abstraction within the Kernel, was created to initialize the databases of the Inner Traffic Controller. This initialization includes the creation of: 1) the Processor Data Segment (PRDS), 2) an MMU Map, 3) Kernel domain stack segments for Kernel processes, 4) allocation and updating of MMU entries for Kernel processes, and 5) Virtual Processor Table (VPT) entries.

The PRDS was loaded with constant values that specify the physical CPU ID, logical CPU ID, and number of VP's allocated to the CPU. A design decision was made to allocate logical CPU ID's in increments of two (beginning with zero) so that they could be used to directly access lists indexed by CPU number. The MMU map, constructed as a "byte" map, was created to specify allocated and free MMU Image entries.

A separate procedure, CREATE_STACK, was created to establish the initial Kernel domain stack conditions for Kernel processes. A discussion and diagram of these initial stack conditions is presented in the next section. Map and allocates the next ALLOCATE MMU checks the MMU availabe MMU entry to the process being created. is inserted in the allocated MMU entry and the DBR number is returned to the calling procedure. The DBR number (handle) is merely the offset of the DBR in the MMU Image. Since the ITC deals with an address rather than a handle, a procedure, GET_DBR_ADDR, was created to convert this offset into a physical address. UPDATE_MMU_IMAGE is the procedure which creates or modifies MMU Image entries. UPDATE_MMU_IMAGE accepts as arguments the DBR number, segment number, segment attributes, and segment limits. To facilitate switching and control, various process segments must possess the same segment number system wide. This is accomplished initialization through the use of the UPDATE_MMU_IMAGE procedure. In the ITC, these segments include the PRDS (segment number zero) and the Kernel stack segment (segment number one).

The final task required in ITC intialization is the creation of the VPT. The VPT header is initialized with the "running" and "ready" lists pointers set to a 'nil' state, and the "free" list pointer set to the first entry in the message table. Virtual Processor entries are inserted in the main body of the VPT by the UPDATE_VP_TABLE procedure. Entries are first made for the VP's permanently bound to the Memory Manager and Ille processes. The VP bound to the MM process is given a priority of 2 (highest), and the VP bound to the Idle process is given a priority of 0 (lowest). The External VP ID for both of these VP's is set to "nil" as they are not visible to the Traffic Controller. The remaining VP's allocated to the CPU (viz., TC visible VP's) are then entered in the VPT with a priority of 1 (intermediate), and their "idle" and "preempt" flags are set. The preempt flag is set for these TC visible VP's to insure proper scheduling by the Traffic Controller. The DBR for these remaining VP's is initialized with the Idle process DBR. A discussion of "idle" processes and VP's will be provided later in this chapter. The External VP ID for each TC visible VP is merely the offset of the next available entry in the EXTERNAL VP LIST. This External VP ID is entered in the VPT, and the corresponding VP ID (viz., VPT Entry #) is entered in the EXTERNAL VP LIST.

Once these VPT entries have been made, it is necessary to set the state of each VP to "ready" and thread them (by priority) into the appropriate ready list. A VPT threading mechanism was provided by Reitz [12] in procedure MAKE_READY. However, it was desired to have a more general threading mechanism that could be used for other lists. Procedure LIST_INSERT was created to provide this general threading mechanism. LIST_INSERT is logically a "library" function that exists at the lowest level of abstraction in the Kernel. This function threads an object into a list (specified by the caller) in order of priority, and then sets its state as specified by the calling parameters.

Once the "Bootstrap Loader" has completed ITC initialization, it passes control to the ITC GETWORK procedure to begin VP scheduling.

2. Traffic Controller Initialization

The initialization routines for the TC include TC_INIT, CREATE_PROCESS, and CREATE_KST. These routines are called from the Memory Manager process. The MM process was chosen to initiate these routines as it is bound to the highest priority VP and will begin running immediately after the Inner Traffic Controller is initialized. Procedure MM_ALLOCATE was written to allocate memory space for data structures during initialization (viz., Kernel stacks, user stacks, and KST's). Memory space is allocated in blocks of

100 (hex) bytes. MM_ALLOCATE is merely a stub of the memory allocating procedure designed by Moore and Gary [5].

It was necessary to pass long lists of arguments to the TC for initialization purposes. To aid in this passing of parameters, a data structure template was used. This template was created by declaring the parameters as a data structure in both the sending and receiving procedures, and then imaging this structure at absolute address zero. The process' stack pointer was then decremented by the size of the parameter data structure, and the parameters were loaded into this data structure indexed by the stack pointer. This template made it very easy to send and receive long argument lists using the process' stack segment.

TC_INIT initializes the APT header and virtual interrupt vector (discussed later). Each element of the running list is marked "idle", the ready and blocked lists are set to "nil", and the number of VP's and first VP for each CPU are entered in the VP table. The address of the virtual preempt handler is then passed to the ITC procedure CREATE_INT_VEC for insertion in the virtual interrupt vector.

CREATE_PROCESS intializes user processes and creates entries in the APT. ALLOCATE_MMU is called to acquire a DBR number, and an APT entry is created with the process descriptors (viz., parameters). The process is then declared "ready" and threaded into the approciate ready list by calling the threading function, LIST_ENSERT. A user stack

is allocated and UPDATE_MMU_IMAGE is called to include the user stack in the MMU as segment number three. The user stack contains no information or user process initialization parameters (viz., execution point and address space) as all processes are initialized and begin execution from the Kernel domain. Next, a Kernel domain stack is allocated and included in the MMU Image. A design decision was made to initialize the Kernel stacks for user processes with the same structure as the Kernel process' stacks. The rationale for this decision is presented in the next section. As a result of this decision, it became possible to use the CREATE_STACK procedure in building Kernel domain stacks for both Kernel and user prosesses. CREATE_STACK was therefore used as a library function and placed in the library module with LIST-INSERT.

Finally, a Known Segment Table (KST) stub is created to provide a means of demonstrating the mechanism provided by the eventcounts and sequencers for interprocess communication (IPC) and mutual exclusion. Space for the process' KST is created by calling MM_ALLOCATE. The KST is then included in the process' address space, as segment number two, by UPDATE_MMU_IMAGE. Initial entries are made in the Known Segment Table by procedure CREATE_KST. CREATE_KST makes an entry in the KST for the "root" and marks the remaining KST entries as "available." The Unique_ID portion of the root's handle (viz., upper two words) is initialized as -1 (for

convenience) and the G_AST entry number portion of the handle (viz., lowest word) is initialized with zero.

3. Additional Initialization Requirements

As already mentioned, the Memory Manager Process prepares the arguments utilized by TC_INIT, CREATE_PROCESS, CREATE_KST for TC initialization and user process creation. Additionally, the MM process creates a Global Active Segment Table (G_AST) stub utilized for demonstration of event data management. The G_AST stub is declared in a separate module (viz., the DEMO_DATABASE Module) with the format prescribed by Moore and Gary [5]. However, the only fields initialized utilized by this implementation are UNIQUE_ID, and SEQUENCER, INSTANCE 1, and INSTANCE 2. The eventcounts and sequencer fields are initialized as zero whenever an entry is created in the G_AST. The UNIQUE_ID is created just to support this demonstration and does not reflect the segment's unique identifier as specified by Moore and Gary [5]. In this demonstration, UNIQUE ID is built with the parameters passed to MM_ACTIVATE. The first word in UNIQUE_ID is the G_AST entry number of the segment's parent, and the second word is the segment's entry number into the alias table. The UNIQUE ID together with the offset of the segment's entry in the G_AST comprise the segment HANDLE maintained in the KST. The first entry in the G_AST is reserved for the root, and is initialized with an Unique_ID of minus one

(-1). It should be noted that any call to MM_ACTIVATE for a segment already possessing an entry in the G_AST will not effect any changes to that entry. This is to insure that a single G_AST entry exists for every segment as specified by Moore and Gary [5].

B. PREEMPT INTERRUPTS

Various refinements were made in the handling of both physical (hardware) and virtual (software) preempt interrupts. A hardware preempt is a non-vectored interrupt that invokes the virtual processor scheduling mechanism (viz., ITC GETWORK). A virtual preempt is a software vectored interrupt that invokes the user process scheduling mechanism (viz., TC_GETWORK). This implementation provides the notion of a virtual interrupt that closely mirrors the behavior of a hardware interrupt. In particular, there are similar constructs for initialization of a handler, invokation of a handler, masking of interrupts, and return from a handler. As with most hardware interrupts, a virtual interrupt can occur only at the completion of execution for an "instruction," where each kernel entry and exit for a process delimit a single "virtual instruction."

1. Physical Preempt Handler

The physical preempt handler resides in the virtual processor manager (viz., Inner Traffic Controller). The functions it perform are: 1) save the execution point, 2) invoke ITC GETWORK, 3) check for virtual preempt interrupts, 4) restore the execution point, and 5) return control via the IRET instruction. Reitz [12] included the hardware preempt handler in ITC GETWORK by establishing two entry points and two exit points, one for a regular call to GETWORK and another for the preempt interrupt. He had a separate procedure, TEST_PREEMPT, that was used to check for the occurrence of virtual preempt interrupts. This structure works nicely, but it requires some means of determining how GETWORK was invoked so that the proper exiting mechanism is used. This was resolved by incorporating a preempt interrupt flag in the status register block of every process. Kernel domain stack segment. A design decision was made to restructure the hardware preempt handler into a single and separate procedure, PHYS_PREEMPT_HANDLER. This allowed ITC GETWORK to have a single entry and exit point, and it did away with the necessity of maintaining a preempt interrupt flag in the process stacks. PHYS_PREEMPT_HANDLER was constructed from the preempt handling code in GETWORK procedure TEST_PREEMPT. TEST_PREEMPT was deleted from the ITC as its functions were performed by PHYS_PREEMPT-HANDLER.

A further refinement was made to the hardware preempt handler dealing with the method by which the virtual preempt handler was invoked. Reitz [12] invoked the virtual preempt handler from TEST_PREEMPT by means of the "call" instruction. Since the virtual preempt handler logically exists at a higher level of abstraction than the ITC, this invocation violated our notion of only allowing "calls" to lower or equal abstraction levels. However, this deviation was necessitated by the absence of a virtual interrupt structure. This problem was alleviated by creating a virtual interrupt vector in the ITC that is used in the same way as the hardware interrupt vector. The virtual preempt was given a virtual interrupt number (zero). The virtual interrupt handler is then invoked by means of a "jump" through the virtual interrupt vector for virtual interrupt number 0. This invocation occurs in the same manner that the handlers for hardware interrupts are invoked. The virtual interrupt vector is created by procedure CREATE_INT_VEC. CREATE_INT_VEC accepts as arguments a virtual interrupt number and the address of the interrupt handler. The creation of the virtual preempt entry in the virtual interrupt vector is accomplished at the time of the Traffic Controller initialization by TC_INIT.

2. Virtual Preempt Handler

The virtual preempt handler (VIRT_PREEMPT_HANDLER) resides in the user process manager (viz., the Traffic Controller). The functions performed by VIRT_PREEMPT_HANDLER are: 1) determine the VP ID of the virtual processor being preempted, 2) invoke the process scheduling mechanism (viz., TC_GETWORK), and 3) return control via a virtual interrupt return. The correct VP ID is obtained by calling RUNNING_VP in the ITC. The Active Process Table is then locked, and the state of the process running on that VP is changed to "ready." At this time, process scheduling is effected by calling TC_GETWORK. Once process scheduling is completed, the APT is unlocked and control is returned via a virtual interrupt return. This virtual interrupt return is merely a jump to the PREEMPT_RET label in the hardware preempt hanaler (This jump emulates the action of the IRET instruction for a hardware interrupt return). This label is the point at which the virtual preempt interrupts are unmasked.

All Kernel processes are initialized to appear as though they are returning from a hardware preempt interrupt. All user processes initially appear to be returning from a virtual preempt interrupt. Therefore, the initial conditions of a process' Kernel domain stack is largely influenced by the stack manipulation of the preempt handlers. Figure 44 illustrates the initial Kernel domain stack structure for all system processes.

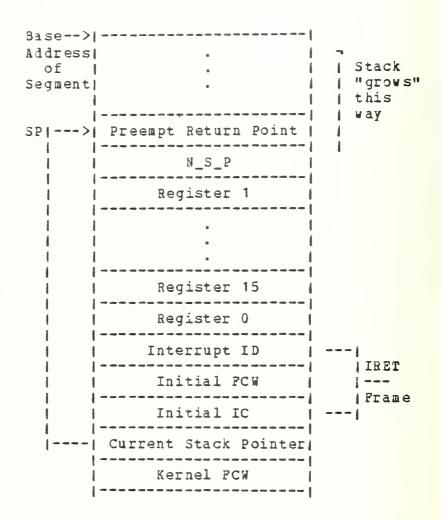


Figure 44: Initial Process Stack

The initial Kernel Flag Control Word (FCW) value is "5000", indicating non-segmented code, system mode of operation, non-vectored interrupts masked, and vectored interrupts enabled. The Current Stack Pointer value is set to the first entry in the stack (viz., SP). The IRET Frame is the portion of the Kernel stack affected by the IRET instruction. The first element, Interrupt ID (set to "FFFF") is merely popped off of the stack and discarded. The next element, Initial FCW, is popped and placed in the system Flag Control Word. Initial FCW is set to "5000" for Kernel processes and "1800" (indicating normal mode with all interrupts enabled) for user processes. The final element of the IRET frame, Initial IC is popped off of the stack and placed in the program counter (PC) register. This value is initialized as the entry address of the process in question.

The "register" entries on the stack represent the initial register contents for the process at the beginning of its execution. Since the Kernel processes (viz., MM and Idle) do not require any specific initial register states, their entries reflect the register contents at the time of stack creation. Initial register conditions are used to provide initial "parameters" required by the user processes. This will depend largely upon the parameter passing conventions of the implementation language. The means for register initialization was provided through CREATE_PROCESS; however, the only initial register condition used for the user

processes in this demonstration was register #13. #13 was used to pass the user ID/Host number of the process created. This value is utilized by the user process in activating the segment used for inter-process communication between a Host's File manager and I/O processes. Another logical parameter passed to the user processes is the root segment number. This did not require a register for passing in the demonstration as it is known to be the first entry in the KST for all processes. The N_S_P entry on the stack represents the initial value of the normal stack pointer. For user processes, this value is obtained when the Supervisor domain stack for that process is created. For Kernel processes, this value is set to "FFFF" since they execute solely in the Kernel domain and have no Superivsor domain stack. The Preempt Return Point specifies the address where control will be passed once the process' VP is scheduled and the "return" from ITC GETWORK is executed. For Kernel processes, this is the point within the hardware preempt handler where the virtual processor table is unlocked. user processes, this is the point within the virtual preempt handler where the Active Process Table is unlocked.

It is important to note that if the APT was not unlocked when a user process began its initial execution, the system would become deadlocked and no further process scheduling could occur. It should be further noted that the initial stack conditions for user processes do not reflect a valid

history of execution. The "normal" history of a user process returning from ITC GETWORK after a virtual preempt interrupt would reflect the passing of control through SWAP VDBR and TC_GETWORK to the point in the virtual preempt handler where the APT is unlocked. Another "possible" history could reflect the occurrence of a hardware preempt interrupt at the point in the virtual preempt handler where the APT is unlocked. Such a history would be depicted by replacing the current top of the stack with the return point into the hardware preempt handler (viz., at the point of virtual preempt interrupt unmasking) and an additional hardware preempt interrupt frame whose IC value in the IRET frame is the point in the virtual preempt handler where the APT is unlocked. The current initial stack condition for user processes was chosen for its ease of understanding and its clear depiction of the fact that the structure of a Kernel domain stack is the same for both Kernel and user processes.

C. IDLE PROCESSES

In the SASS design, there logically exists a Kernel domain "Idle" process for every physical processor in the system and a Supervisor domain "Idle" process for every "TC visible" virtual processor in the system. These processes are necessary to insure that both the VP scheduler (viz., ITC GETWORK) and the process scheduler (TC_GETWORK) will always

have some object to schedule, hence precluding any CPU or VP from ever having an undefined execution point. Since the Kernel domain Idle process performs no useful work, it could be included within the ITC by means of an infinite looping mechanism. The Kernel Idle process was maintained separately, however, as it is hoped that future work on SASS will provide this Idle process with some constructive purpose (e.g., performing maintenance diagnostics).

The Supervisor domain Idle processes (hereafter referred to as TC Idle processes) are scheduled (bound) on VP's when there are no user processes awaiting scheduling. Since a TC Idle process performs no user constructive work, we do not want any VP executing a TC Idle process to be bound to a physical processor. In other words, a VP bound to a TC Idle process assumes the lowest system priority (represented by the "idle flag"). Therefore, any such VP will have its idle flag set and will not be scheduled unless it receives a virtual preempt interrupt. Such an interrupt will allow the VP to be rescheduled by the Traffic Controller. It should be obvious, at this point, that a TC Idle process will never This knowactually begin execution on a real processor. ledge allowed a design decision to be made to only simulate the existence of TC Idle processes. At the TC level, this was accomplished by a constant value, IDLE PROC, that was used as a process ID in the APT running list, thus precluding the necessity of any "Idle" entries in the APT. At the

ITC level, any VP marked "Idle" (viz., the idle flag set) was given the DBR number (viz., address space) of the Kernel Idle process solely to provide the use of a Kernel domain stack for rescheduling of the VP.

D. ADDITIONAL KERNEL REPINEMENTS

In addition to those already discussed, several other refinements to existing Kernel modules were effected in this implementation. One of these refinements deals with the way virtual processors are identified by the Traffic Controller. In the current implementation, all TC visible virtual processors are given an External VP ID which corresponds to its entry number in an External VP List. This required a modification to the ITC procedure RUNNING_VP. The benefits derived from this refinement included the ability to directly access the External VP ID in the Virtual Processor Table vice the requirement of a run time division to compute its value and the ability to use the External VP ID as an index into the TC running list.

Refinements were also made to the existing Memory Manager, File Manager and IO process stubs used for demonstration purposes. These refinements were largely associated with the eventcount and sequencer mechanisms utilized in this implementation. The current status of these processes is provided in this report.

The remaining refinements deal largely with the MMU Image. In Moore and Gary's [5] design, the MMU Image was managed by the Memory Manager process. This was largely because the MMU Image is a processor local database and would seem well suited for management by the non-distributed Kernel. In fact, the MMU Image is utilized mainly by the ITC for the multiplexing of process address spaces. Therefore, in the current design, the MMU Images are maintained by the Inner Traffic Controller. However, the MMU header proposed by Moore and Gary (viz., the BLOCKS_USED and MAXIMUM_AVAILABLE_BLOCKS fields) was retained in the Memory Manager as it is used strictly in the management of a process' virtual core and is not associated with the hardware MMU.

In Wells' design [20], the Traffic Controller used the linear ordering of the DBR entries in the MMU Image as the DBR handle (viz., 1,2,3...). This required a run time division operation to compute the DBR number, and a run time multiplication operation, by MM_GET_DBR_VALUE, to recompute the DBR address for use by the ITC. In the current design, the offset of the DBR entry in the MMU Image (obtained at the time of MMU allocation) is used as the DBR handle in the Traffic Controller. Furthermore, SWAP_VDBR was refined to accept a DBR handle rather than a DBR address to preclude the necessity of the Traffic Controller having to deal with MMU addresses. DBR addresses are computed only within the

ITC (viz., by procedure GET_DBR_ADDR) by adding the value of the DBR handle to the base address of the MMU Image. Since DBR addresses are now used solely within the ITC, procedure MM_GET_DBR_VALUE was no longer needed and was deleted from the Memory Manager.

E. SUMMARY

The primary issues addressed in this thesis effort have been presented in this chapter. Aside from the process management functions, this description included a mechanism for limited Kernel database initialization, a revised preempt interrupt handling mechanism, the creation of a virtual interrupt structure, a definition of "idle" processes and their structure, and a discussion of the minor refinements effected in existing SASS modules. A detailed description of the implementation of process management functions for the SASS is presented in the next chapter.

Chapter XXII

PROCESS MANAGEMENT IMPLEMENTATION

The implementation of process management functions and a gate keeper stub (system trap) is presented in this chapter. The implementation is discussed in terms of the Event Manager, Traffic Controller, Distributed Memory Manager, User Gate, and Kernel Gate Keeper modules. A block diagram depicting the structure and interrelationships of these modules is presented in Figure 45. Support in developing the Z8000 machine code for this implementation was provided by a Zilog MCZ Developmental System operating under the RIO operating The Developmental System provided disk file management for a dual drive, hard sectored floppy disk, a line oriented text editor, a PLZ/ASM assembler, a linker and a loader that created an executable image of each 28000 load upload/download capability with the Am96/4116 MonoBoard computer was also provided. This capability, along with the general interfacing of the Am96/4116 into the SASS system, was accomplished in a concurrent thesis endeavor by Gary Baker. Baker's work relating to hardware initialization in SASS, will be published upon completion of his thesis work in June 1981.

Gate Keeper	Kernel Gate	
 Read Await	Ticket	Advance
	Convert and Verify Event Manager	
TC_Await	 Process Class	TC_Advance
TC_Getwork	•	
Traffic Controller		
MM_Read_Eventcount	MM_Ticket	MM_Advance
Distributed Memory Manager		

Figure 45: Implementation Module Structure

A. EVENT MANAGER MODULE

The eventcount and sequencer primitives [11], which are system-wide objects, collectively comprise the event data of SASS. As mentioned earlier, this event data is tied directly to system segments and is stored in the Global Active Segment Table. There are two eventcounts and one sequencer for every segment in the system. These objects are identified to the Kernel in user calls by specification of a segment number. Once this segment number is identified by the Kernel, the segment's handle can be obtained from the process' Known Segment Table. The segment handle identifies the particular entry in the G_AST containing the event data desired.

The Event Manager module manages the event data within the system and provides the mechanism for interprocess communication between user processes. The Event Manager consists of six procedures. Four of these (Advance, Await, Read, and Ticket) represent the four user extended instructions provided by the Event Manager. The remaining two procedures provide internal computational support to include necessary security checking. The Event Manager is invoked solely by user processes, via the Gate Keeper, through utilization of the extended instruction set provided. For every Event Manager extended instruction invoked by a user process, the non-discretionary security is verified by comprocess, the non-discretionary security is verified by com-

paring the security access classification of the process invoking the instruction with the classification of the object (segment) being accessed. Access to the user process Known Segment Table is required by the module in order to ascertain the segment handle and security class for a given segment number. The PLZ/ASM assembly language listing for the Event Manager module is provided in Appendix A. A more detailed discussion of the procedures comprising the Event Manager follows.

1. Support Procedures

The procedures GET_HANDLE and CONVERT_AND_VERIFY provide internal support for the Event Manager and are not visible to the user processes. Procedure CONVERT AND VERIFY is invoked by the four procedures representing the instruction set of the Event Manager. The input parameters to CONVERT_LNC_VERIFY are a segment number and a requested mode of access (viz., read or write). CCNVERT_AND_VERIFY returns a pointer to the segment's handle and a success code. GET_HANDLE is invoked solely Procedure by CONVERT AND VERIFY. The input parameter to GET_HANDLE is the segment number received as input by CONVERT_AND_VERIFY. GET_HANDLE returns a pointer to the segment's handle, a pointer to the segment's security classification, and a success code. A discussion of the functions provided by these support procedures follows.

Procedure GET_HANDLE translates the segment number, ceived as input, into a KST index number and verifies that the resulting index number is valid. Next the base address process' KST is obtained from procedure οf the KST index number is then converted ITC_GET_SEG_PTR. The into a KST offset value and added to the base address to obtain the appropriate KST entry pointer for the segment in question. A verification is then made to insure that the referenced segment is "known" to the process. If the segan error message is returned to ment is not known, CONVERT_AND_VERIFY. Otherwise, a pointer to the segment's handle is obtained to identify the segment to the memory manager. A pointer to the segment's security class entry in the KST is also returned for use in appropriate security checks.

Procedure CONVERT_AND_VERIFY provides the necessary non-discretionary security verification for the extended instruction set of the Event Manager. Procedure GET_HANDLE is invoked for segment number verification and to obtain pointers to the segment's handle and security class. If GET_HANDLE returns with a successful verification, the process' security class is compared to the segment's security class to verify the mode of access requested. A request for "write" access causes invocation of the CLASS_EQ function in the Non-Discretionary Security Module to insure that the security classification of the process is equal to the classi-

fication of the eventcount or sequencer, which is the same as that of the segment. Otherwise, the CLASS_GE function is called to verify that the process has read access. If the appropriate security check is unsuccessful, an error code is returned by CONVERT_AND_VERIFY. Otherwise, the segment handle is returned along with a success code of "succeeded" indicating that the user process possesses the necessary security clearance to complete execution of the extended instruction.

2. Read

Procedure READ ascertains the current value of a user specified eventcount and returns its value to the caller. The input parameters to READ are a segment number and an instance (viz., an event number). CONVERT_AND_VERIFY is invaked with a "read" access request to obtain the segment's handle and necessary verification. "Read" access is sufficient for this operation as it only requires observation of the current eventcount value and performs no data modification. If verification is successful, procedure MM_READ_EVENTCOUNT is called to obtain the eventcount value.

3. Ticket

Procedure TICKET returns the current sequencer value for the segment specified by the user. CONVERT_AND_VERIFY is called with a request for write access to obtain verifica-

tion and the segment handle. Write access is required because once the sequencer value is read it must be incremented in anticipation of the next ticket request. Once verification is complete, MM_TICKET is invoked to obtain the sequencer value that is returned to the user process. It is noted that every call to TICKET for a particular segment number will return a unique and time ordered sequencer value. This is because the sequencer value may only be read within MM_TICKET while the G_AST is locked, thereby preventing simultaneous read operations. Puthermore, once the sequencer value is read it is incremented before the G_AST is unlocked.

4. Await

Procedure AWAIT allows a user process to block itself until some specified event has occurred. The parameters to AWAIT include a segment number and instance, which identify a particular event, and a user specified value which identifies a particular occurrence of the event. Verification of read access and a pointer to the segment's handle is obtained from procedure CONVERT_AND_VERIFY. Procedure TC_AWAIT is invoked to effect the actual waiting for the event occurrence. TC_AWAIT will not return to AWAIT until the requested event has occurred. It is noted that AWAIT makes no assumptions about the event value specified by the user. Therefore, the Kernel cannot guarantee that the event

specified by the user will ever occur; this is the responsibility of other cooperating user processes.

5. Advance

Procedure ADVANCE allows a user process to broadcast the occurrence of some event. This is accomplished by incrementing the value of the eventcount associated with the event that has occurred. The parameters to ADVANCE include a segment number and instance which identify a particular event. The calling process must have write access to the identified segment as modification of the eventcount is required. Verification of write access and a pointer to the segment's handle is obtained through procedure CONVERT_AND_VERIFY. Procedure TC_ADVANCE is invoked to perform the actual broadcasting of event occurrence.

E. TRAFFIC CONTROLLER MODULE

The primary functions of the Traffic Controller module are user process scheduling and support of the inter-process communication mechanism. The Traffic Controller is invoked by the occurrence of a virtual preempt interrupt and by the Event Manager and the Segment Manager through the extended instruction set: TC_Advance, TC_Await, Process_Class, and Get_DBR_NUMBER. The Traffic Controller module is comprised of nine procedures. Four of these procedures represent the extended instruction set of the Traffic Controller. A de-

tailed discussion of six of the procedures contained in the Traffic Controller module is presented below. The remaining three procedures (viz., TC_INIT, CREATE_PROCESS, and CREATE_KST) were described in chapter three. The PLZ/ASM assembly language source code listings for the Traffic Controller module is provided in Appendix B.

1. TC Getwork

Procedure TC_GETWORK provides the mechanism for user process scheduling. The input parameters to TC_GETWORK are the VP ID of the virtual processor to which a process will be scheduled and the logical CPU number to which the virtual processor belongs. The determination of which process to schedule is made by a looping mechanism that finds the first "ready" process on the ready list associated with the current logical CPU number. Processes appear in the ready list by order of priority. This looping mechanism is required as both "running" and "ready" processes are maintained on the ready list. This ready list structure was chosen to simplify the algorithm provided in procedure TC_Advance. ready process is found, its state is changed to "running" and its process ID (viz., the APT entry number) is inserted in the running list entry associated with the current virtual processor. Procedure SWAP VDBR is then invoked in the Inner Traffic Controller to effect the actual process switch. If a ready process was not found (viz., the ready

list was empty or comprised solely of "running processes"), then the running list entry associated with the current virtual processor is marked with the constant "Idle_Proc" and procedure IDLE is invoked in the Inner Traffic Controller.

2. TC Await

The primary function of TC_AWAIT is the determination of whether some user specified event has occurred. If the event has occured, control is returned to the caller. Otherwise, the process is blocked and another process is scheduled. The input parameters to TC_AWAIT are a pointer to a segment handle, an instance (event number), and a user specified eventcount value. TC_AWAIT initially locks the Active Process Table and obtains the current value of the evin question by calling procedure entcount MM_READ_EVENTCOUNT. The determination of event occurrence is made by comparing the user specified eventcount value with the current eventcount. If the user value is less than or equal to the current eventcount, the awaited event has occurred and control is returned to the caller. Otherwise, the awaited event has not yet occurred and the process must be blocked.

If the process is to be blocked, procedure RUNNING_VP is invoked to ascertain the VP ID of the virtual processor bound to the process. The process' ID (viz., APT entry number) is then read from the running list. The input parame-

then stored in the Event Data portion of the process APT entry. The process is removed from its associated ready list by redirecting the appropriate linking threads (pointers). Once removed from the ready list, the process is threaded into the blocked list and its state changed to "blocked" by invocation of the library function LIST_INSERT. Procedure TC_GETWORK is then called to schedule another process for the current virtual processor.

3. TC Advance

The primary purpose of TC_ADVANCE is the broadcasting of some event occurrence. This entails incrementing the eventcount associated with the event, awakening all processes that are waiting for the event, and insuring proper scheduling order by generating any necessary virtual preempt inter-The high level design algorithm for TC_ADVANCE is rupts. provided in Figure 46. The input parameters to TC ADVANCE are a pointer to a segment's handle and an instance (event number). Initially, TC_ADVANCE locks the APT to prevent the possibility of a race condition. The eventcount identified by the input parameters is then incremented by calling MM_ADVANCE. MM_ADVANCE returns the new value of the eventcount. Once the eventcount has been advanced, TC ADVANCE awakens all processes awaiting this event occurrence. is accomplished by checking all processes that are currently in the blocked list. The process' HANDLE and INSTANCE entries are compared with the handle and instance identifying the current event. If they are the same, then the process is awaiting some occurrence of the current event. In such a case, the process' VALUE entry in the APT is compared with the current value of the eventcount. If the process' VALUE is less than or equal to the current eventcount value, the awaited event has occurred and the process is removed from the blocked list and threaded into the appropriate ready list by the library function LIST_INSERT.

Once the blocked list has been checked, it is necessary to reevaluate each ready list to insure that the highest priority processes are running. It is relatively simple to determine if a virtual preempt interrupt is necessary, however, it is considerably more difficult to determine which virtual processor should receive the virtual preempt. assist in this evaluation, a "count" variable (number of on the Kernel stack with an entry for every virtual processor associated with the logical CPU being evaluated. tially, every entry in the preempt vector is marked "true" indicating that its associated virtual processor is a candidate for preemption. Once the preempt vector is initialized, the first "n" processes on the ready list (where n equals the number of VP's associated with the current logical CPU) are checked for a determination of their state. If

```
TC ADVANCE Procedure (HANDLE, INSTANCE)
Begin
  ! Get new eventcount !
 COUNT := MM ADVANCE (HANDLE, INSTANCE)
 Call WAIT_LOCK (APT)
  ! Wake up processes !
 PROCESS := BLOCKED_LIST_HEAD
 Do while not end of BLOCKED LIST
    If (PROCESS.HANDLE = HANDLE) and
       (PROCESS.INSTANCE = INSTANCE) and
       (PROCESS.COUNT <= COUNT)
    then
       Call LIST INSERT (READY LIST)
   end if
   PROCESS := PROCESS.NEXT_PROCESS
 end do
 ! Check all ready lists for preempts !
 LOGICAL_CPU_NO := 1
 Do while LOGICAL_CPU_NO <= #NR_CPU
   ! Initialize preempt vector !
   VP_ID := FIRST VP(LOGICAL CPU NO)
   Do for LOOP := 1 to NR_VP(LOGICAL_CPU NO
     RUNNING_LIST[VP_ID].PREEMPT := #TRUE
     VP_ID := VP_ID + 1
   end do
   ! Find preempt candidates !
   CANDIDATES := 0
   PROCESS := READY_LIST_HEAD (LOGICAL_CPU NO)
```

Figure 46: TC ADVANCE Algorithm

```
VP_ID := FIRST_VP(LOGICAL_CPU_NO)
   Do (for CYCLE = 1 to NR_VP(LOGICAL_CPU_NO) and
      not end of READY_LIST(LOGICAL_CPU_NO)
     If PROCESS = #RUNNING
       then
        RUNNING_LIST[VP_ID].PREEMPT := #FALSE
       else
        CANDIDATES := CANDIDATES + 1
     end if
     VP_ID := VP_ID + 1
     PROCESS := PROCESS.NEXT_PROCESS
   end do
   ! Preempt appropriate candidates !
   VP_ID := FIRST_VP(LOGICAL_CPU_NO)
   Do for CHECK := 1 to NR_VP(LOGICAL_CPU_NO)
      If (RUNNING_LIST[VP_ID].PREEMPT = #TRUE) and
         (CANDIDATES > G)
       then
       Call SET PREEMPT (VP_ID)
        CANDIDATES := CANDIDATES - 1
     end if
     VP_ID := VP_ID + 1
   end do
   LOGICAL_CPU_NO := LOGICAL_CPU_NO + 1
 end do
 Call UNLOCK (APT)
 Return
End IC_ADVANCE
```

Figure 46: TC_ADVANCE Algorithm (Continued)

a process is found to be "running" then it should not be preempted as processes appear in the ready list in order of priority. When a running process is found, its associated entry in the preempt vector is marked "false." If a process is encountered in the "ready" state then it should be running and the "count" variable is incremented. When the first "n" processes have been checked or when we reach the end of the current ready list (whichever comes first), the entries in the preempt vector are "popped" from the stack. If an entry from the preempt vector is found to be "true", this indicates that its associated virtual processor is a candidate for preemption since it is either bound to a lower priority process, or it is "idle." In such a case, the "count" variable is evaluated to determine if the virtual processor associated with the vector entry should preempted. If the count exceeds zero, a virtual preempt interrupt is sent to the VP and the count is decremented. Otherwise, no preempt is sent as there is no higher priority process awaiting scheduling.

This preemption algorithm is completed for every ready list in the Active Process Table. Once all ready lists have been evaluated, the APT is unlocked and control is returned to the caller. It is noted that it is not necessary to invoke TC_GETWORK before exiting ADVANCE. If the current VP requires rescheduling, it will have received a virtual preempt interrupt from the preemption algorithm. If this

has occurred, the VP will be rescheduled when its running process attempts to leave the Kernel domain and the virtual preempt interrupts are unmasked.

4. Virtual Preempt Handler

VIRTUAL_PREEMPT_HANDLER is the interrupt handler for virtual preempt interrupts. The entry address of VIRTUAL_PREEMPT_HANDLER is maintained in the virtual interrupt vector located in the Inner Traffic Controller. Once invoked, the handler locks the Active Process Table and determines which virtual processor is being preempted by calling RUNNING_VP. The process running on the preempted VP is then set to the "ready" state and TC_GETWORK is invoked to reschedule the virtual processor. When TC_GETWORK returns to VIRTUAL_PREEMPT_HANDLER, the APT is unlocked and a virtual interrupt return is executed. This return is simply a jump to the point in the hardware preempt handler where the virtual interrupts are unmasked. This effects a virtual interrupt return instruction.

5. Remaining Procedures

The remaining two procedures in the Traffic Controller module represent the extended instructions: PROCESS_CLASS and GET_DBR_NUMBER. Both procedures lock the Active Process Table and call RUNNING_VP to determine which virtual processor is executing the current process. The process ID (viz.,

APT entry Number; is then extracted from the running list. PROCESS_CLASS reads and returns the current process' security access classification from the APT. GET_DBR_NUMBER reads and returns the current process' DBR handle. It should be noted that in general the DBR number provided by procedure GET_DBR_NUMBER is only valid while the APT is locked. Particularly, in the current SASS implementation, the Segment Manager invokes GET_DBR_NUMBER and then passes the obtained DBR number to the Distributed Memory Manager for utilization at that level. In a more general situation, the process associated with the DBR number may have been unloaded before the DBR number was utilized, thus making it invalid. This problem does not arise in SASS as all processes remain loaded for the life of the system.

C. DISTRIBUTED MEMORY MANAGER MODULE

The Distributed Memory Manager module provides an interface between the Segment Manager and the Memory Manager process, manipulates event data in the Global Active Segment Table (G_AST), and dynamically allocates available memory. A detailed description of the Distributed Memory Manager interface to the Memory Manager process was presented by Wells [20]. The remaining extended instruction set is discussed in detail below. The complete PLZ/ASM source listings for the Distributed Memory Manager module is provided in Appendix C.

1. MM Read Eventcount

MM_READ_EVENTCOUNT is invoked by the Event Manager and the Traffic Controller to obtain the current value of the eventcount associated with a particular event. The input parameters to this precedure are a segment handle pointer and an instance (event Number), which together uniquely identify a particular event.

The G_AST is locked and the entry offset of the segment into the G_AST is obtained from the segment's handle. The instance parameter is then validated to determine which eventcount is to be read. If an invalid instance is specified, control is returned to the caller specifying an error condition. Otherwise, the current value of the specified eventcount is read. The G_AST is then unlocked, and the current eventcount value is returned to the caller.

2. MM_Advance

MM_ADVANCE is invoked by the Traffic Controller to reflect the occurrence of some event. The input parameters to MM_ADVANCE are a pointer to a segment's handle and a particular instance (event number).

The Global Active Segment Table is locked to prevent a race condition, and the offset of the segment's entry into the G_AST is obtained from the segment handle. The instance parameter is then validated to determine which eventcount is to be advanced. If an invalid instance is specified, an er-

ror condition is returned to the caller and no data entries are affected. If the instance value is valid, the appropriate eventcount is incremented, and its new value is returned.

3. MM_Ticket

MM_TICKET is invoked by the Event Manager to obtain the current value of the sequencer associated with a specified segment. The input parameter to MM_TICKET is a pointer to a segment's handle.

Initially, MM_TICKET locks the Global Active Segment Table to prevent a race condition. Next the offset of the segment's entry into the G_AST is obtained from the segment handle. The current value of the sequencer for the specified segment is then read and saved as a return parameter to the caller. The sequencer value is then incremented in anticipation of the next ticket request. Once this is complete, the G_AST is unlocked and control is returned to the caller.

4. MM_Allocate

The MM_ALLOCATE procedure provided in this implementation is a stub of the MM_ALLOCATE described in the Memory Manager design of Moore and Gary [5]. The primary function of MM_ALLOCATE is the dynamic allocation of fixed size blocks of available memory space. It is invoked in the cur-

rent implementation by the initialization routines in BOOTSTRAP_LOADER and TC_INIT for the allocation of memory space used in the creation of the Kernel domain and Supervisor domain stack segments and the creation of the Known Segment Tables for user processes. Dynamic reallocation of previously used memory space (viz., garbage collection) is not provided by the MM_ALLOCATE stub in this implementation. All memory allocation required in this implementation is for segments supporting system processes that remain active, and thus allocated, for the entire life of the system. Memory is allocated in blocks of 256 (decimal) bytes of processor local memory (on-board RAM). In this stub allocatable memory is declared at compile time by a data structure (MEM_POOL) that is accessible only by MM_ALLOCATE.

The input parameter to MM_ALLOCATE is the number of blocks of requested memory. This parameter is converted from a block size to the actual number of bytes requested. This computation is made simple since memory is allocated in powers of two. The byte size is obtained by logically shifting left the input parameter eight times, where eight is the power of two desired (viz., 256). Once the size of the requested memory is computed, it is necessary to determine the starting address of the memory block (s) to be allocated. To assist in this computation, a variable (NEXT_BLOCK) is used to keep track of the next available block of memory in MEM_POOL. NEXT_BLOCK, which is initial—

ized as zero, provides the offset into the memory being allocated. Once the starting address is obtained, the physical size of the memory allocated is added to NEXT_BLOCK so that the next request for memory allocation will begin at the next free byte of memory in MEM_POOL. This new value of NEXT_BLOCK is saved and the starting address of the memory for this request is returned to the caller.

D. GATE KEEPER MODULES

The SASS Gate Keeper provides the logical boundary between the Supervisor and the Kernel and isolates the Kernel from the system users, thus making it tamperproof. This is accomplished by means of the hardware system/normal mode and the software ring-crossing mechanism provided by the Gate Keeper. The Gate Keeper is comprised of two separate modules: 1) the USER_GATE module, and 2) the KERNEL_GATE_KEEPER module. These modules are disjoint, with the USER_GATE module residing in the Supervisor domain and the KERNEL_GATE_KEEPER module residing in the Kernel domain. It is important to note that the USER GATE is a separately linked component in the Supervisor domain and is not linked to the Kernel. The only thing in common between these two modules is a set of constants identifying the valid extended instruction set which the Kernel provides to the users.

The Gate Keeper modules presented in this implementation are only stubs as they do not provide all of the functions

required of the Gate Keeper. However, the only task not provided in this implementation is the validation of parameters passed from the Supervisor to the Kernel. A detailed description of this parameter validation design is provided by Coleman [2]. In the process management demonstration, the Supervisor stubs are written in PLZ/ASM with all parameters passed by CPU registers. A detailed description of the Gate Keeper modules and the nature of their interfaces is presented below. The PLZ/ASM source listings for the two Gate Keeper modules are provided in Appendix D.

1. User Gate Module

The USER_GATE module provides the interface structure between the user processes in the Supervisor domain and the Kernel. The USER_GATE is comprised of ten procedures (viz., entry points) that correlate on a one to one basis with the ten "user visible" extended instructions (listed in Figure 10) provided by the Kernel. The only action performed by each of these procedures is the execution of the "system call" instruction (SC) with a constant value, identifying the particular extended instruction invoked, as the source operand.

The SC instruction is a system trap that forces the hardware into the system mode (Kernel domain) and loads register 15 with the system stack pointer (Kernel domain stack). The current instruction counter value (IC) is

pushed onto the Kernel stack along with the current CPU flag control word (FCW). In addition, the system trap instruction is pushed onto the Kernel stack with the upper byte representing the SC instruction and the lower byte representing the SC instruction's source operand (viz., the Kernel extended instruction code). Together, these operations form an interrupt return (IRET) frame as illustrated in Pigure 44. Once this is complete, the FCW is loaded with the FCW value found in the System Call frame of the Program Status Area (viz., the hardware "interrupt vector"). The structure of the Program Status Area is illustrated in Figure 47. The instruction counter is then loaded with the address of the SC instruction trap handler. This value is also located in the SC frame of the Program Status Area.

OFFSET

```
Reserved | |--Frames
   Unimplemented |
     Instruction
   Trap
8
   | Privileged
  Institution Trap
      Instruction |
12
  Kernel PCW | System
   | Kernel Gate Keeper | | Instruction
  Address
16
Segment | Trap |
   Non-Maskable | Interrupt |
24 |----- ---- --- Hardware
  | Kernel FCW | Preempt | PHYS_PREEMPT_HANDLER | (Non-Address | Vectored
28 |----| --| Interrupt)
   | Vectored Int |
32
```

* NOTE: Offsets represent Program Status Area structure for non-segmented Z8002 microprocessor.

Figure 47: Program Status Area

2. Kernel Gate Keeper Module

The system trap handler for the System Call instruction address of the KERNEL GATE KEEPER. The the is KERNEL_GATE_KEEPER and the Kernel FCW value are placed in the System Call frame of the Program Status Area by the during initialization. BOOTSTRAP LOADER module The KERNEL_GATE_KEEPER fetches the extended instruction code from the trap instruction entry in the IRET frame on the Kernel stack. This value is then decoded by a "case" statement to determine which extended instruction is to be executed. If the extended instruction code is valid, the appropriate Kernel procedure is invoked. Otherwise, an error condition is set and no Kernel procedures are not invoked. Once control returns to the KERNEL_GATE_KEEPER, the CPU registers and normal stack pointer (NSP) value are pushed onto the Kernel stack in preparation for return to the Supervisor It is noted that this operation would normally ocdomain. cur immediately upon entry into the KERNEL GATE KEEPER. In this implementation, however, parameter validation is not accomplished and the CPU registers are used to pass parameters to and from the Kernel only for use by the process management demonstration. In an actual SASS environment, all parameters would be passed in a separate argument list and the CPU registers would appear exactly the same upon leaving the Kernel as they did upon entering the Kernel. This is

important to insure that no data or information is leaked from the Kernel by means of the CPU registers.

Control is returned to the Supervisor by means of the return mechanism in the hardware preempt handler. This mechanism is utilized to preclude the necessity of building a separate mechanism for the KERNEL_GATE_KEEPER that would actually perform the very same function. To accomplish this, the KERNEL_GATE_KEEPER executes an unconditional jump to the PREEMPT_RET label in PHYS_PREEMPT_HANDLER. This "jump" to the hardware preempt handler represents a "virtual IRET" instruction providing the same function as the virtual interrupt return described in the discussion of the virtual preempt handler. At this point, the virtual preempt interrupts are unmasked, the normal stack pointer and CPU registers are restored from the stack, and control is returned to the Supervisor by execution of the IRET instruction.

E. SUMMARY

The implementation of process management functions for the SASS has been presented in this chapter. The implementation was discussed in terms of the Event Manager, Traffic Controller, Distributed Memory Manager, and Gate Keeper modules.

Chapter XXIII

CONCLUSION

The implementation of process management for the security Kernel of a secure archival storage system has been presented. The process management functions presented provide a logical and efficient means of process creation, control, and scheduling. In addition, a simple but effective mechanism for inter-process communication, based on the eventcount and sequencer primitives, was created. Work was also completed in the area of Kernel database initialization and a Gate Keeper stub to allow for dual domain operation.

The design for this implementation was based on the Zilog Z8001 sixteen bit segmented microprocessor [22] used in
conjunction with the Zilog Z8010 Memory Management Unit
[23]. The actual implementation of process management for
the SASS was conducted on the Advanced Micro Computers
Am96/4116 MonoBoard Computer [1] featuring the AmZ8002 sixteen bit non-segmented microprocessor. Segmentation hardware was simulated by a software Memory Management Unit Image.

This implementation was effected specifically to support the Secure Archival Storage System (SASS) [17]. However, the implementation is based on a family of Operating Systems

[7] designed with a primary goal of providing multilevel information security. The loop free modular design utilized in this implementation easily facilitates any required expansion or modification for other family members. In addition, this implementation fully supports a multiprocessor design. While the process management implementation appears to perform correctly, it has not been subjected to a formal test plan. Such a test plan should be developed and implemented before kernel verification is begun.

A. FOLLOW ON WORK

There are several possible areas in the SASS design that would be immediately suitable for continued research. In the area of hardware, this includes, the establishment of a multiprocessor environment, hardware initialization, and interfacing to the host computers and secondary storage. Further work in the Kernel includes the actual implementation of the memory manager process, and the refinement of the Gate Keeper and Kernel intialization structures. The implementation of the Supervisor has not been addressed to date. Its areas of research include the implementation of the File Manager and Input/Output processes, and the final design and implementation of the SASS-Hosts protocols.

Other areas that could also prove interesting in relation to the SASS include the implementation of dynamic memory management, the support of multilevel hosts, dynamic pro-

cess creation and deletion, and the provision of constructive work to be performed by the Idle process.

Appendix A

EVENT MANAGER LISTINGS

```
Z8000ASM 2.02
LOC OBJ CODE
                  STMT SOURCE STATEMENT
       $LISTON $TTY
       EVENT MGR MODULE
       CONSTANT
         TRUE
                              := 1
         FALSE
                              := 0
         READ_ACCESS
                              := 1
         WRITE ACCESS
                             := 0
         SUCCEEDED
                              := 2
         SEGMENT_NOT_KNOWN
                             := 28
         ACCESS_CLASS_NOT_EQ := 33
         ACCESS_CLASS_NOT_GE := 41
         KST_SEG_NO
                             := 2
         NR_OF_KSEGS
                             := 10
         MAX_NO_KST_ENTRIES
                             := 54
         NOT_KNOWN
                              := %FF
       TYPE
         H_ARRAY
                       ARRAY[3 WORD]
         KST_REC RECORD
          [MM HANDLE
                      H ARRAY
           SIZE
                      WORD
           ACCESS_MODE BYTE
           IN CORE BYTE
           CLASS
                      LONG
           M_SEG_NO SHORT_INTEGER
           ENTRY_NUMBER SHORT_INTEGER]
       EXTERNAL
         MM TICKET
                             PROCEDURE
```

MM_READ_EVENTCOUNT

PROCESS_CLASS

TC ADVANCE

TC_AWAIT

CLASS_EQ CLASS GE PROCEDURE

PROCEDURE

PROCEDURE

PROCEDURE PROCEDURE

PROCEDURE

INTERNAL

SSECTION EM_KST_DCL
! NOTE: THIS SECTION IS AN "OVERLAY"
OR "FRAME" USED TO DEFINE THE
FORMAT OF THE KST. NO STORAGE IS
ASSIGNED BUT RATHER THE KST IS
STORED IN A SEPARATELY OBTAINED
AREA. (A SEGMENT SET ASIDE FOR IT)!

\$ABS 0

0000 KST ARRAY[MAX_NO_KST_ENTRIES KST_REC]

GLOBAL \$SECTION EM_GLB_PROC

```
0000
                               PROCEDURE
          · **********************
           * READS SPECIFIED EVENTCOUNT *
           * AND RETURNS IT'S VALUE TO
           * THE CALLER
           *********
           * PARAMETERS:
             R1: SEGMENT #
              R2: INSTANCE
           ********
           * RETURNS:
             RO: SUCCESS CODE
              RR4: EVENTCOUNT
           ************
           ENTRY
            ! SAVE INSTANCE !
0000 93F2
            PUSH
                  aR15, R2
            ! "READ" ACCESS REQUIRED !
0002 2102
            LD
                  R2, #READ_ACCESS
3004 0001
            ! GET SEG HANDLE & VERIFY ACCESS !
0006 5F00
            CALL CONVERT AND VERIFY !R1:SEG #
0000 8000
                                      R2: REQ. ACCESS
                                      RETURNS:
                                      RO:SUCCESS CODE
                                      R1: HANDLE PTR!
          CP RO, #SUCCEEDED
000A 0B00
000C 0002
            IF EQ ! ACCESS PERMITTED!
000E 5E0E
            THEN ! READ EVENTCOUNT!
0010 001C
              !RESTORE INSTANCE!
0012 97F2
             POP R2, aR15
0014 5F00
             CALL MM_READ_EVENTCOUNT !R1:HPTR
0016 0000*
                                      R2: INSTANCE
                                      RETURNS:
                                      RO:SUCCESS CODE
                                      RR4: EVENTCOUNT!
0018 5E08
            ELSE ! RESTORE SP!
001A 001E'
             POP R2. @R15
001C 97F2
            FI
001E 9E08
           RET
           END READ
0020
```

```
0020
                              PROCEDURE
          TICKET
          * RETURNS CURRENT VALUE OF
          * TICKET TO CALLER AND INCRE- *
          * MENTS SEQUENCER FOR NEXT
          * TICKET OPERATION
          **********
          * PARAMETERS:
          * R1: SEGMENT #
          **********
          * RETURNS:
          * RO: SUCCESS CODE
             RR4: TICKET VALUE
          ***********
          ENTRY
           ! GET SEG HANDLE & VERIFY ACCESS !
           ! "WRITE" ACCESS REQUIRED !
0020 2102
          LD R2, #WRITE_ACCESS
0022 0000
0024 5F00
          CALL CONVERT_AND_VERIFY !R1:SEG #
0026 00001
                                  R2: ACCESS REQ
                                  RETURNS:
                                  RO:SUCCESS CODE
                                  R1:HANDLE PTR!
0028 0B00
          CP RO, #SUCCEEDED
002A 0002
           IF EQ ! ACCESS PERMITTED!
002C 5E0E
           THEN ! GET TICKET !
002E 0038*
0030 5700
            CALL MM_TICKET !R1: HANDLE PTR
0032 0000*
                            RETURNS:
                            RR4: TICKET!
             ! RSTORE SUCCESS CODE !
0034 2100
             LD RO. #SUCCEEDED
0036 0002
           FI
0038 9E08
           RET
003A
         END TICKET
```

```
003A
           AWAIT
                               PROCEDURE
          * CURRENT EVENTCOUNT VALUE IS *
           * COMPARED TO USER SPECIFIED
           * VALUE. IF USER VALUE IS
           * GREATER THAN CURRENT EVENT- *
           * COUNT VALUE THEN PROCESS IS *
           * "BLOCKED" UNTIL THE DESIRED *
           * EVENT OCCURS.
           ************
           * PARAMETERS:
              R1: SEGMENT #
              R2: INSTANCE (EVENT #)
              RR4: SPECIFIED VALUE
           ***********
           * RETURNS:
             RO: SUCCESS CODE
           *************
           ENTRY
            ! SAVE DESIRED EVENTCOUNT VALUE !
003A 91F4
            PUSHL @R15, RR4
            ! SAVE INSTANCE !
003C 93F2
            PUSH @R15, R2
            ! "READ" ACCESS REQUIRED !
003E 2102
            LD
                  R2, #READ ACCESS
0040 0001
            ! GET SEG HANDLE & VERIFY ACCESS !
0042 5F00
            CALL CONVERT_AND_VERIFY !R1:SEG #
0044 00001
                                     R2:ACCESS REQ
                                     RETURNS:
                                     RO:SUCCESS CODE
                                     R1:HANDLE PTR!
0046 0B00
                 RO. #SUCCEEDED
           CP
0048 0002
            IF EQ ! ACCESS PERMITTED !
            THEN ! AWAIT EVENT OCCURRENCE !
004A 5E0E
004C 005A'
              ! RESTORE INSTANCE !
004E 97F2
              POP R2, aR15
              ! RESTORE SPECIFIED VALUE !
             POPL RR4, DR15
0050 95F4
0052 5F00
             CALL TC AWAIT !R1: HANDLE PTR
0054 0000*
                            R2: INSTANCE
                            RR4: VALUE
                            RETURNS:
                            RO: SUCCESS CODE!
```

0056 5E08 ELSE !RESTORE STACK!

0058 005E'

005A 95F4 POPL RR4, @R15

005C 97F2 POP R2, @R15

FI

005E 9E08 RET

0060 END AWAIT

```
0060
           ADVANCE
                               PROCEDURE
          * SIGNALS THE OCCURRENCE OF
           * SOME EVENT. EVENTCOUNT IS
           * INCREMENTED AND THE TRAFFIC *
           * CONTROLLER IS INVOKED TO
           * AWAKEN ANY PROCESS AWAITING *
           * THE OCCURRENCE.
           *****************
           * PARAMETERS:
             R1: SEGMENT #
              R2: INSTANCE (EVENT #)
           **********
           * RETURNS:
              RO: SUCCESS CODE
           ***********
           ENTRY
            ! SAVE INSTANCE !
            PUSH DR15, R2
0060 93F2
            ! GET SEG HANDLE & VERIFY ACCESS !
            ! "WRITE" ACCESS REQUIRED !
                 R2, #WRITE_ACCESS
0062 2102
            LD
0064 0000
            CALL CONVERT_AND_VERIFY !R1:SEG #
0066 5F00
0068 0000
                                     R2: ACCESS REQ
                                     RETURNS:
                                     RO:SUCCESS CODE
                                     R1:HANDLE PTR!
           CP RO, #SUCCEEDED
006A 0B00
006C 0002
            IF EQ ! ACCESS PERMITTED !
             THEN ! ADVANCED EVENTCOUNT !
006E 5E0E
0070 007C*
              ! RESTORE INSTANCE !
                   R2. @R15
              POP
0072 97F2
              CALL TC_ADVANCE ! R1: HANDLE PTR
0074 5F00
0076 0000*
                               R2: INSTANCE
                               RETURNS:
                               RO:SUCCESS CODE!
            ELSE ! RESTORE STACK!
0078 5E08
007A 007E
                   R2, @R15
              POP
007C 97F2
             PI
007E 9E08
            RET
            END ADVANCE
0080
```

INTERNAL SSECTION EM_INT_PROC

0000	CONVERT_AND_VERIFY PROCEDURE !************************ * CONVERTS SEGMENT NUMBER TO KST INDEX* * AND EXTRACTS SEGMENT'S HANDLE FROM * * KST. IF SUCCESSFUL, THEN ACCESS * * CLASS OF SUBJECT IS CHECKED AGAINST * * ACCESS CLASS OF OBJECT TO INSURE * * THAT ACCESS IS PERMITTED. * **********************************
	* PARAMETERS: * * R1: SEGMENT NUMBER *
	* R2: ACCESS REQUESTED * ***********************************

	* RO: SUCCESS CODE *
	* R1: HANDLE POINTER * * ********************************

	ENT RY
	! SAVE REQUESTED ACCESS !
0000 93F2	PUSH @R15, R2
COO2 5800	! GET SEGMENT HANDLE ! CALL GET_HANDLE !R1:SEG #
0004 0062	CALL GLI_HANDED : R1.516 F
_	RETURNS:
	RO:SUCCESS CODE
	R4:HANDLE PTR R5:CLASS PTR!
CO06 0B00	CP RO, #SUCCEEDED
0008 0002	
0000 5707	IF EQ ! SEGMENT IS KNOWN!
000A 5E0E 000C 005E	
0000 0031	! SAVE HANDLE & CLASS PTR !
000E 91F4	PUSHL DR 15, RR 4
0010 5F00	! GET SUBJECT'S SAC !
0010 3200	CALL PROCESS_CLASS ! RETURNS:
	RR2:PROC CLASS!
004" 0570	! RETRIEVE SEG CLASS POINTER !
0014 95F0	POPL RRO, DR15 ! GET SEGMENT'S CLASS !
0016 1414	LDL RR4, DR1
	! RETRIEVE REQUESTED ACCESS !
0018 97F1	POP R1, DR15
001A 93F0	! SAVE HANDLE POINTER ! PUSH @R15, RO
33.11 3310	! CHECK ACCESS CLEARANCE !
001C 0801	CP R1, #WRITE_ACCESS
001E 0000	TP PO 1 UDIME LOCACO DEGUARDO .
	IF EQ ! WRITE ACCESS REQUESTED !

```
0022 0040
0024 5F00
                 CALL CLASS_EQ ! RR2: PROCESS CLASS
0026 0000*
                                  RR4: SEGMENT CLASS
                                  RETURNS:
                                  R1: CONDITION CODE!
0028 0B01
                 CP R1, #FALSE
002A 0000
                 IF EQ ! ACCESS NOT PERMITTED!
002C 5E0E
                  THEN
002E 0038'
0030 2100
                   LD
                        RO, #ACCESS_CLASS_NOT_EQ
0032 0021
0034 5E08
                 ELSE !ACCESS PERMITTED!
0036 003C
0038 2100
                       RO, #SUCCEEDED
                  LD
003A 0002
                 FI
003C 5E08
                ELSE ! READ ACCESS REQUESTED !
003E 0058'
0040 5F00
                 CALL CLASS GE !RR2:PROCESS CLASS
0042 0000*
                                  RR4:SEGMENT CLASS
                                  RETURNS:
                                  R1: CONDITION CODE!
0044 OB01
                 CP
                       R1, #FALSE
0046 0000
                 IF EO !ACCESS NOT PERMITTED!
0048 5E0E
                  THEN
004A 00541
004C 2100
                        RO, #ACCESS_CLASS_NOT_GE
                  LD
004E 0029
                 ELSE !ACCESS PERMITTED!
0050 5E08
0052 0058
0054 2100
                  LD RO, #SUCCEEDED
0056 0002
                 FI
               FI
               ! RETRIEVE HANDLE POINTER !
0058 97F1
               POP
                    R1, @R15
005A 5E08
             ELSE
005C 0060'
               ! RESTORE STACK !
               POP R2. aR15
005E 97F2
             PI
0060 9E08
             RET
            END CONVERT_AND_VERIFY
0062
```

0020 5E0E

THEN

```
0062
           GET HANDLE
                                PROCEDURE
          * CONVERTS SEGMENT NUMBER TO
           * KST INDEX AND DETERMINES IF *
           * SEGMENT IS KNOWN. IF KNOWN *
           * POINTER TO SEGMENT HANDLE
           * AND POINTER TO SEGMENT CLASS*
           * ARE RETURNED.
           ************
           * PARAMETERS:
             R1: SEGMENT NUMBER
           ***********
           * RETURNS:
             RO: SUCCESS CODE
             R4: HANDLE POINTER
              R5: CLASS POINTER
           ****************
           ENTRY
            ! CONVERT SEGMENT # TO KST INDEX # !
0062 0301
            SUB
                 R1, #NR_OF_KSEGS
0064 000A
            ! VERIFY KST INDEX !
0066 2100
           LD RO. #SUCCEEDED
0068 0002
006A 0B01
                 R1, #0
            CP
006C 0000
            IF LE ! INDEX NEGATIVE!
006E 5E0A
             THEN
0070 007A1
0072 2100
             LD
                    RO, #SEGMENT NOT KNOWN
0074 001C
0076 5E08
             ELSE ! INDEX POSITIVE!
0078 00861
007A 0B01
              CP
                    R1, #MAX_NO_KST_ENTRIES
007C 0036
              IF GT ! EXCEEDS MAXIMUM INDEX!
007E 5E02
               THEN !INVALID INDEX!
0080 00861
0082 2100
                LD
                    RO, #SEGMENT NOT KNOWN
0084 001C
              FI
            FI
0086 0B00
                 RO, #SUCCEEDED
            CP
0088 0002
            IF EQ
                  !INDEX VALID!
008A 5E0E
             THEN
008C 00BE*
              ! SAVE KST INDEX !
008E 93F1
              PUSH
                   DR 15, R1
              ! GET KST ADDRESS !
0090 2101
              LD
                    R1, #KST_SEG_NO
0092 0002
0094 5F00
              CALL ITC_GET_SEG_PTR !R1: KST_SEG_NO
```

```
0096 0000*
                                      RETURNS:
                                      RO: KST ADDR!
               ! RETRIEVE KST INDEX # !
0098 97F3
              POP R3, aR15
               ! CONVERT KST INDEX # TO KST OFFSET !
009A 1902
               MULT RR2, #SIZEOF KST_REC
009C 0010
               ! COMPUTE KST ENTRY ADDRESS !
               ADD R3, RO
009E 8103
               ! SEE IF SEGMENT IS KNOWN !
00A0 4D31
               CP KST.M_SEG_NO(R3), #NOT_KNOWN
00A2 000E
OOA4 OOFF
               IF EQ !SEGMENT NOT KNOWN!
00A6 5E0E
               THEN
00A8 00B21
00AA 2100
                LD RO, #SEGMENT_NOT_KNOWN
00AC 001C
00AE 5E08
               ELSE !SEGMENT KNOWN!
00B0 00BE .
00B2 2100
                LD RO, #SUCCEEDED
00B4 0002
                 ! GET HANDLE POINTER !
00B6 7634
                 LDA R4, KST. MM_HANDLE (R3)
00B8 0000
                 ! GET CLASS POINTER !
00BA 7635
                LDA R5, KST. CLASS (R3)
00BC 000A
               FI
             FI
00BE 9E08
             RET
            END GET_HANDLE
00C0
           END EVENT_MGR
```

Appendix B

TRAFFIC CONTROLLER LISTINGS

```
Z8000ASM 2.02
LOC OBJ CODE
                 STMT SOURCE STATEMENT
       $LISTON $TTY
       TC MODULE
        CONSTANT
        ! ****** SYSTEM PARAMETERS ****** !
               NR_PROC
                              := 4
               VP_NR
                               := 2
               NR_CPU
                              := 2
               NR_KST
                               := 54
         ! ****** SYSTEM CONSTANTS ****** !
              RUNNING
                          := 0
                          := 1
              READY
              BLOCKED
                          := 2
              IDLE_PROC
                          := %D DD D
                          := %FFFF
              NIL
              INVALID
                          := %EEEE
              KERNEL_STACK := 1
              USER_STACK
                         := 3
              KST_SEG
                          := 2
              KST LIMIT
                          := 1
              USER FCW
                          := %1800
                          := 0
              WRITE
              !INDICATES LOWEST SYSTEM
              SECURITY CLASS!
              SYSTEM_LOW := 0
              STK OFFSET
                          := %PF
              REMOVED
                          := %ABCD
                          := 1
              TRUE
              FALSE
                          := 0
              SUCCEEDED
                          := 2
        TYPE
         AP PTR
                     WORD
```

WORD

WORD

VF_PTR

ADDRESS

HARRAY

ARRAY[3 WORD]

```
AP_TABLE RECORD
       [NEXT_AP
                       AP PTR
        DBR
                       WORD
        SAC
                       LONG
        PRI
                       INTEGER
        STATE
                       INTEGER
        AFFINITY
                       WORD
        VP_ID
                       VP PTR
        HANDLE
                       H_ARRAY
        INSTANCE
                       WORD
        VALUE
                        LONG
        FILL 2
                       ARRAY[2 WORD]
 RUN ARRAY
              ARRAY[VP_NR AP_PTR]
 RDY_ARRAY
               ARRAY[NR_CPU AP_PTR]
 AP_DATA
               ARRAY[NR_PROC AP_TABLE]
 VP_DATA
               RECORD
   [NR_VP
               ARRAY[NR CPU WORD]
   FIRST
               ARRAY[NR_CPU VP_PTR]
   1
 KST_REC
               RECORD
             H_ARRAY
  [MM_HANDLE
   SIZE
               WORD
   ACCESS
               BYTE
   IN_CORE
               BYTE
   CLASS
               LONG
   M_SEG_NO SHORT_INTEGER ENTRY_NUM SHORT_INTEGER
EXTERNAL
   K LOCK
                       PROCEDURE
   K_UNLOCK
                       PROCEDURE
   SET PREEMPT
                       PROCEDURE
   SWAP_VDBR
                       PROCEDURE
   IDLE
                       PROCEDURE
   RUNNING_VP
                       PROCEDURE
   CREATE_INT_VEC
                      PROCEDURE
   LIST_INSERT
                       PROCEDURE
   ALLOCATE_MMU
                      PROCEDURE
   MM_ALLOCATE
                       PROCEDURE
   UPDATE_MMU_IMAGE
                      PROCEDURE
   CREATE STACK
                       PROCEDURE
   MM_ADVANCE
                       PROCEDURE
   MM READ EVENTCOUNT PROCEDURE
   G AST LOCK
                       WORD
   PREEMPT_RET
                       LABEL
```

```
SSECTION TC_DATA
         INTERNAL
0000
          APT
                     RECORD
            LOCK
                               WORD
              RUNNING LIST
                               RUN ARRAY
              READY_LIST
                               RDY_ARRAY
              BLOCKED_LIST
                               AP PTR
              FILL_3
                               LONG
                               VP DATA
              V P
              FILL
                               ARRAY[4 WORD]
              AP
                               AP DATA
             1
         !THESE VARIABLES ARE USED DURING TO
          INITIALIZATION TO SPECIFY AVAILABLE
          ENTRIES IN THE APT, AND ARE INITIAL-
          IZED BY TC_INIT IN THIS IMPLEMENTATION!
00A0
         NEXT_VP
                     WORD
00A2
         APT_ENTRY WORD
        $SECTION TC_LOCAL
        $ABS 0
        !NOTE: USED AS OVERLAY ONLY!
0000
         ARG LIST RECORD
           [ R EG
                        ARRAY[ 13 WORD ]
            IC
                         WORD
            CPU_ID
                         WORD
            SAC1
                         LONG
            PRI1
                         WORD
            USR_STK
                         WORD
            KER STK
                         WORD
            KST1
                         LONG
           1
        $ABS 0
        INOTE: USED AS STACK FRAME FOR
         STORAGE OF TEMPORARY VARIABLES
         FOR CREATE_PROCESS.!
0000
         CREATE
                    RECORD
           [ARG_PTR
                      WORD
            DBR NUM
                      WORD
            LIMITS
                      WORD
            SEG ADDR ADDRESS
            N S P
                      WORD
           1
        $ABS 0
0000
         HANDLE_VAL
                      RECORD
             [ HIGH
                    LONG
              LOW
                    WORD
             ]
        !THE FOLLOWING DECLARATION IS UTILIZED
         AS A STACK FRAME FOR STORAGE OF
```

```
TEMPORARY VARIABLES UTILIZED BY
         TC_ADVANCE AND TC_AWAIT.!
        $ABS 0
         TEMP
0000
                 RECORD
           [HANDLE_PTR
                          WORD
            EVENT NR
                          WORD
            EVENT_VAL
                          LONG
            ID_Ab
                          WORD
            CPU NUM
                          WORD
            HANDLE_HIGH
                          LONG
            HANDLE_LOW
                          WORD
           1
        $SECTION TC_KST_DCL
         !NOTE: KST DECLARATION IS USED HERE
          TO SUPPORT KST INITIALIZATION FOR
          THIS DEMONSTRATION ONLY.
                                    THIS
          DECLARATION AND INITIALIZATION
          SHOULD EXIST AT THE SEGMENT MANAGER
          LEVEL AND THUS SHOULD BE REMOVED
          UPON IMPLEMENTATION OF SYSTEM
          INITIALIZATION.!
            $ABS 0
0000
             KST ARRAY[NR_KST KST_REC]
```

```
SSECTION TO INT PROC
0000
                               PROCEDURE
           TC GETWORK
          · *************************
           * PROVIDES GENERAL MANAGE-
           * MENT OF USER PROCESSES BY *
           * EFFECTING PROCESS SCHEDU- *
           * LING ON VIRTUAL PROCESSORS*
           **********
           * PARAMETERS:
             R1: CURRENT VP ID
              R3: LOGICAL CPU #
           **********
           * LOCAL VARIABLES:
             R2: NEXT READY PROCESS
             R4: AP PTR
           ****************
           ENTRY
            ! FIND FIRST READY PROCESS !
0000 6132
            LD
                 R2. APT.READY LIST (R3)
0002 0006
            GET_READY_AP:
            DO
                !WHILE NOT (END OF LIST OR READY)!
0004 0B02
                  R2, #NIL
            CP
0006 FFFF
0008 5E0E
             IF EQ !NO READY PROCESS! THEN
000A 0010'
000C 5E08
             EXIT FROM GET READY AP
000E 0026
             FI
0010 4D21
             CP
                 APT. AP. STATE (R2) , #READY
0012 002A1
0014 0001
0016 5E0E
            IF EQ !PROCESS READY! THEN
0018 001E.
001A 5E08
             EXIT FROM GET READY AP
001C 00261
             PI
             ! GET NEXT AP FROM LIST !
001E 6124
             LD
                 R4, APT.AP.NEXT AP(R2)
0020 0020
0022 A142
                 R2, R4
             LD
0024 E8EF
            OD
0026 0B02
            CP
                 R2,#NIL
0028 FFFF
002A 5E0E
            IF EQ ! IF NO PROCESSES READY! THEN
002C 003C1
             ! LOAD IDLE PROCESS !
002E 4D15
             LD
                 APT.RUNNING_LIST(R1), #IDLE_PROC
0030 0002
0032 DDDD
0034 5F00
            CALL IDLE
0036 0000*
0038 5E08 ELSE
```

```
003A 0052*
              ! LOAD FIRST READY AP !
003C 6F12
              LD APT.RUNNING_LIST(R1), R2
003E 0002
              LD APT. AP. STATE (R2), #RUNNING
0040 4D25
0042 002A
0044 0000
                  APT.AP. VP_ID(R2), R1
0046 6F21
             LD
0048 002E
004A 6121
             LD R1, APT.AP.DBR (R2)
004C 00221
004E 5F00
             CALL SWAP_VDBR ! (R1: DBR)!
0050 0000*
             FI
             RET
0052 9E08
           END TC_GETWORK
0054
```

```
VIRTUAL PREEMPT_HANDLER
0054
                                   PROCEDURE
           · ************************
            * LOADS FIRST READY AP
            * IN RESPONSE TO PREEMPT
            * INTERRUPT
            ***********
            ENTRY
             !** CALL WAIT LOCK (APT-.LOCK) **!
             !** RETURNS WHEN PROCESS HAS LOCKED APT **!
0054 7604
             LDA R4, APT.LOCK
0056 0000
0058 5F00
            CALL K LOCK
005A 0000*
             ! GET RUNNING VP ID !
005C 5F00
             CALL RUNNING_VP !RETURNS:
005E 0000*
                                R1: VP_ID
                                R3:CPU #!
             ! GET AP !
0060 6112
            LD
                   R2, APT.RUNNING LIST(R1)
0062 0002
             ! IF NOT AN IDLE PROCESS, SET IT TO READY!
0064 0B02
             CP
                   R2. #IDLE PROC
0066 DDDD
            IF NE ! NOT IDLE ! THEN
0068 SE06
006A 3072
006C 4D25
            LD APT.AP.STATE(R2), #READY
006E 002A
0070 0001
             FI
             ! LOAD FIRST READY PROCESS !
0072 5F00
            CALL TC_GETWORK !R1:VP_ID
0074 0000
                               R3:CPU #!
            !NOTE: THIS IS THE INITIAL POINT OF
             EXECUTION FOR USER PROCESSES.!
            VIRT_PREEMPT_RETURN:
             *** CALL UNLOCK (APT-.LOCK) **!
             *** RETURNS WHEN PROCESS HAS UNLOCKED APT **!
             !** AND ADVANCED ON THIS EVENT **!
0076 7604
            LDA R4. APT.LOCK
0078 0000
007A 5F00
            CALL K UNLOCK
007C 0000*
             ! PERFORM A VIRTUAL INTERRUPT RETURN !
             INOTE: THIS JUMP EFFECTS A VIRTUAL
             IRET INSTRUCTION.!
            JP PREEMPT_RET
007E 5E08
*0000 0800
```

```
GLOBAL
        $SECTION TC_GLB_PROC
0000
           TC INIT
                             PROCEDURE
          * INITIALIZES APT HEADER
           * AND VIRTUAL INT VECTOR
           ***********
           * PARAMETERS:
                                    *
             R1: CPU ID
             R2: NR VP
           ***************
           ENTRY
            ! NOTE: THE NEXT FOUR VALUES ARE
             ONLY TO BE INITIALIZED ONCE. !
0000 4D05
                NEXT_VP, #0
            LD
0002 00A0*
0004 0000
0006 4D05
           LD
                APT_ENTRY, #0
0008 00A2*
000A 0000
000C 4D05
           LD
                APT.BLOCKED_LIST, #NIL
000E 000A*
0010 FFFF
0012 4D08
           CLR
                APT.LOCK
0014 0000
            NOTE: THE FOLLOWING CODE IS INCLUDED
            ONLY FOR SIMULATION OF A MULTIPROCESSOR
            ENVIRONMENT.
                         THIS IS TO INSURE THAT THE
            READY LIST(S) AND VP DATA OF THE SIMULATED
            CPU(S) ARE PROPERLY INITIALIZED. IN AN
             ACTUAL MULTIPROCESSOR ENVIRONMENT, THIS
            BLOCK OF CODE SHOULD BE REMOVED.
            0016 2104
                 LD
                      R4, #0
0018 0000
                 DO
001A 0B04
                       R4, #NR CPU*2
                  CP
001C 0004
                  IF EQ !ALL LISTS INITIALIZED!
001E 5E0E
                   THEN EXIT
0020 0026
0022 5E08
0024 00361
                  ! INITIALIZE READY LISTS AS EMPTY !
0026 4D45
                  LD
                       APT.READY LIST (R4), #NIL
0028 0006
002A FFFF
                  ! INITIALLY MARK ALL LOGICAL CPU'S
                    AS HAVING 1 VP. THIS IS NECESSARY
                    TO INSURE TC ADVANCE WILL FUNCTION
                    PROPERLY, AS IT EXPECTS EVERY CPU
```

```
TO HAVE AT LEAST 1 VP. !
002C 4D45
                    LD
                         APT. VP. NR VP(R4), #1
002E 0010*
0030 0001
0032 A941
                         R4, #2
                    INC
0034 E8F2
                   OD
             ! END MULTIPROCESSOR SIMULATION CODE.
              **************
                  APT. VP. NR_VP(R1), R2
0036 6F12
             LD
0038 0010
003A 6103
             LD
                  R3, NEXT VP
003C 00A0*
003E 6F13
             LD
                  APT. VP. FIRST (R1), R3
0040 0014
            ! RECOMPUTE NEXT_VP VALUE FOR TC
              INITIALIZATION OF NEXT LOGICAL
              CPU. !
0042 A125
                  R5, R2
             LD
0044 1904
             MULT RR4, #2
0046 0002
                  R3, R5
0048 8153
             ADD
004A 6F03
                  NEXT_VP, R3
             LD
004C 00A0
             ! INITIALIZE RUNNING LIST !
             LD R3. APT. VP. FIRST (R1)
004E 6113
0050 0014
             DO
             CP R2, #0
0052 0B02
0054 0000
0056 5E0E
             IF EQ THEN EXIT FI
0058 005E*
005A 5E08
005C 006A
              LD APT.RUNNING_LIST (R3), #IDLE_PROC
005E 4D35
0060 0002
0062 DDDD
0064 A931
              INC
                  R3, #2
                  R2, #1
0066 AB20
             DEC
0068 E8F4
             OD
             LD
                  APT.READY LIST(R1), #NIL
006A 4D15
006C 0006'
006E FFFF
                 R1, #0
0070 2101
             LD
0072 0000
             ! ENTRY ADDRESS !
             LDA R2, VIRTUAL PREEMPT_HANDLER
0074 7602
0076 0054
             CALL CREATE_INT_VEC
0078 5F00
007A 0000*
                    !R1:VIRTUAL INTERRUPT #
                     R2: INTERRUPT HANDLER ADDRESS!
007C 9E08
             RET
            END TC_INIT
007E
```

```
CREATE PROCESS PROCEDURE
007E
           ***************
            * CREATES USER PROCESS
            * DATABASES AND APT
            * ENTRIES
            ***********
            * PARAMETERS:
               R14: ARGUMENT PTR
            *************
            ENTRY
             INOTE: THIS PROCEDURE IS A STUB TO ALLOW
              PROCESS INITIALIZATION FOR THIS
              DEMONSTRATION.!
             ! ESTABLISH STACK PRAME FOR LOCAL
               VARIABLES. !
007E 030F
             SUB R 15, #SIZEOF CREATE
A000 0800
             ! STORE INPUT ARGUMENT POINTER !
0082 6FFE
                   CREATE. ARG_PTR (R15), R14
0084 0000
             ! LOCK APT !
0086 7604
             LDA
                  R4, APT.LOCK
00088 0000
008A 5F00
             CALL K LOCK
008C 0000*
             ! RETURNS WHEN APT IS LOCKED !
             ! CREATE MMU ENTRY FOR PROCESS !
             CALL ALLOCATE_MMU ! RETURNS:
008E 5F00
0090 0000*
                                  RO: DBR #1
             ! GET NEXT AVAILABLE ENTRY IN APT !
0092 6101
             LD
                   R1, APT_ENTRY
0094 00A2*
             ! COMPUTE APT OFFSET !
0096 2102
             LD
                  R2, #SIZEOF AP TABLE
0098 0020
009A 8112
             ADD
                  R2, R1
             ! SAVE NEXT AVAILABLE APT ENTRY !
009C 6F02
             LD
                  APT_ENTRY, R2
009E 00A2*
             ! CREATE APT ENTRY FOR PROCESS !
00A0 4D15
             LD
                  APT. AP. NEXT AP (R1), #NIL
00A2 0020*
00A4 FFFF
00A6 6F10
             LD
                   APT. AP. DBR (R1), RO
00A8 0022*
             ! GET PROCESS CLASS !
00AA 54E2
             LDL
                   RR2, ARG_LIST.SAC1(R14)
00AC 001E
00AE 5D12
             LDL
                  APT. AP. SAC (R1), RR2
00B0 0024*
             ! GET PROCESS PRIORITY !
00B2 61E2
             LD
                   R2, ARG_LIST.PRI1(R14)
```

```
00B4 0022
00B6 6F12
              LD APT. AP. PRI (R1), R2
00B8 0028'
              ! GET LOGICAL CPU # !
00BA 61E2
              LD
                    R2, ARG_LIST.CPU_ID(R14)
00BC 001C
00BE 6F12
              LD
                    APT. AP. AFFINITY (R1), R2
00C0 002C1
              !THREAD IN LIST AND MAKE READY!
00C2 7623
                    R3, APT.READY_LIST(R2)
              LDA
00C4 0006'
00C6 7604
              LDA
                    R4, APT.AP.NEXT_AP
00C8 00201
00CA 7605
                    R5. APT. AP. PRI
              LDA
00CC 00284
00CE 7606
                   R6. APT.AP.STATE
              LDA
00D0 002A
                   R7, #READY
00D2 2107
              LD
00D4 0001
00D6 AD21
              EX
                    R1, R2
              ! SAVE DBR # 1
00D8 6FF0
              LD
                    CREATE. DBR_NUM (R15), RO
00DA 0002
00DC 5700
              CALL
                    LIST INSERT
00DE 0000*
                    !R2: OBJ ID
                     R3: LIST HEAD PTR
                     R4: NEXT OBJ PTR
                     R5: PRIORITY PTR
                     R6: STATE PTR
                     R7: STATE!
              ! UNLOCK APT !
                   R4, APT.LOCK
00E0 7604
             LDA
00E2 0000°
00E4 5F00
                   K_UNLOCK
              CALL
00E6 0000*
              !CREATE USER STACK!
              ! RESTORE ARGUMENT POINTER !
00E8 61FE
              LD
                    R14, CREATE. ARG_PTR(R15)
00EA 0000
00EC 61E3
             LD
                    R3, ARG_LIST.USR_STK(R14)
00EE 0024
              ! SAVE LIMITS !
00F0 6FF3
             LD
                   CREATE. LIMITS (R15), R3
00F2 0004
00F4 5F00
             CALL MM ALLOCATE !R3: # OF BLOCKS
00F6 0000*
                                  RETURNS:
                                  R2: START ADDR!
              !COMPUTE & SAVE NSP!
00F8 A128
                    R8, R2
              ! ESTABLISH INITIAL SP VALUE
                FOR USER STACK. !
00FA 0108
                   R8, #STK_OFFSET
             ADD
```

```
OOFC OOFF
OOFE 6FF8
            LD CREATE. N_S_P(R15), R8
0100 0008
             ! RESTORE LIMITS !
0102 6184
            LD
                  R4, CREATE.LIMITS (R15)
0104 0004
                   R4 !SEG LIMITS!
0106 AB40
             DEC
             ! RESTORE DBR !
0108 61F0
             LD
                  RO, CREATE. DER NUM (R15)
010A 0002
010C 2101
            LD R1, #USER_STACK
010E 0003
0110 2103
            LD
                  R3. #WRITE !ATTRIBUTE!
0112 0000
0114 5F00
             CALL UPDATE_MMU_IMAGE
0116 0000*
                   !RO: DBR #
                    R1: SEGMENT #
                    R2: SEG ADDRESS
                    R3: SEG ATTRIBUTES
                    R4: SEG LIMITS!
             !CREATE KERNEL STACK!
             ! RESTORE ARGUMENT POINTER !
0118 61FE
             LD
                   R14, CREATE. ARG_PTR(R15)
011A 0000
011C 61E3
             LD
                   R3, ARG_LIST.KER_STK(R14)
011E 0026
0120 5F00
             CALL MM_ALLOCATE !R3: # OF BLOCKS
0122 0000*
                                 RETURNS
                                 R2: START ADDR!
             IMAKE MMU ENTRY!
             ! RESTORE DBR # !
0124 61F0
             LD
                   RO, CREATE. DBR NUM (R15)
0126 0002
0 1 2 8 2 1 0 1
                R1, #KERNEL_STACK
            LD
012A 0001
012C A134
             LD
                  R4, R3
012E AB40
             DEC
                   R4
0130 2103
             LD
                   R3, #WRITE
0132 0000
             ! SAVE START ADDRESS !
0134 6FF2
            LD
                  CREATE. SEG_ADDR (R15), R2
0136 0006
0138 5F00
             CALL
                  UPDATE_MMU_IMAGE
013A 0000*
                   !RO: DBR #
                    R1: SEGMENT #
                    R2: SEG ADDRESS
                    R3: SEG ATTRIBUTES
                    R4: SEG LIMITS!
             !ESTABLISH ARGUMENTS!
             ! RESTORE ARGUMENT POINTER !
                   R14. CREATE. ARG_PTR(R15)
013C 61FE
            LD
```

```
013E 0000
              ! RESTORE STACK ADDRESS !
0140 61F1
              LD
                    R1, CREATE.SEG_ADDR(R15)
0142 0006
0144 2103
              LD
                    R3, #USER_FCW
0146 1800
0148 61E4
              LD
                    R4, ARG_LIST.IC(R14)
014A 001A
              ! RESTORE INITIAL NSP !
014C 61F5
              LD
                    R5, CREATE.N_S_P (R15)
014E 0008
0150 7606
              LDA
                    R6, VIRT_PREEMPT_RETURN
0152 0076
0154 030F
              SUB
                    R15, #8
0156 0008
0158 1CF9
              LDM
                  aR15, R3, #4
015A 0303
              ! LOAD ARGUMENT POINTER FOR
                CREATE_STACK CALL !
015C A1F0
                    RO, R15
              LD
                    @R15, R1
015E 93F1
              PUSH
0160 A1E1
              LD
                    R1, R14
              ! LOAD INITIAL REGISTER VALUES TO
                BE PASSED TO USER PROCESS AS
                INITIAL PARAMETERS. !
0162 5C11
              LDM
                   R2, ARG_LIST.REG(R1), #13
0164 020C
0166 0000
0168 97F1
             POP
                    R1, DR15
016A 5F00
             CALL CREATE_STACK
016C 0000*
                    ! RO: ARGUMENT PTR
                     R1: TOP OF STACK
                     R2-R14: INITIAL
                      REG STATES!
              !NOTE: THE ABOVE INITIAL REG STATES
               REPRESENT THE INITIAL PARAMETERS
               (VIZ., REGISTER CONTENTS) THAT A
               USER PROCESS WILL RECEIVE UPON
               INITIAL EXECUTION. !
016E 010F
                    R15, #8
                            !OVERLAY PARAMETERS!
              ADD
0170 0008
              ! ALLOCATE KST !
0172 2103
             LD
                   R3, #KST_LIMIT
0174 0001
0176 5F00
             CALL MM ALLOCATE !R3:# OF BLOCKS
0178 0000*
                                  RETURNS
                                  R2: START ADDR!
              ! RESTORE DBR !
017A 61F0
                    RO. CREATE. DBR NUM (R15)
             LD
017C 0002
              ! SAVE KST ADDRESS !
017E 6FF2
             LD
                    CREATE. SEG_ADDR (R15), R2
```

0180 0006	
	!MAKE MMU ENTRY FOR KST SEG!
	LD R1, #KST_SEG
0184 0002	
	LD R3, #WRITE !ATTRIBUTE!
0188 0000	
	LD R4, #KST_LIMIT-1
018C 0000	
	CALL UPDATE_MMU_IMAGE
0190 0000*	
	!RO: DBR #
	R1: SEGMENT #
	R2: SEG ADDRESS
	R3: SEG ATTRIBUTES
	R4: SEG LIMITS!
	! RESTORE KST ADDRESS !
	LD R2, CREATE.SEG_ADDR(R15)
0194 0006	
	! CREATE INITIAL KST STUB !
	CALL CREATE_KST !R2:KST ADDR!
0198 01A0'	
	! REMOVE TEMPORARY VARIABLE
	STACK FRAME. !
	ADD R15, #SIZEOF CREATE
019C 000A	
019E 9E08	
0 1 A O	END CREATE_PROCESS

```
CREATE_KST
01A0
                          PROCEDURE
           * CREATES KST STUB FOR *
            * PROCESS MANAGEMENT
            * DEMO. INSERTS ROOT
            * ENTRY IN KST.
                            NOT
            * INTENDED TO BE FINAL *
            * PRODUCT.
            **************
            * PARAMETERS:
              R2: KST ADDRESS
                                  *
            *********
            ENTRY
             !NOTE: THIS PROCEDURE IS A STUB USED
              FOR INITIALIZATION IN THIS IMPLEMENTATION
                    THE ACTUAL INITIALIZATION CODE
              FOR THE KST WILL RESIDE AT THE SEGMENT
              MANAGER LEVEL ONCE IMPLEMENTATION OF
              SYSTEM INITIALIZATION IS EFFECTED. !
             ! CREATE ROOT ENTRY IN KST !
01A0 1406
             LDL
                  RR6, #-1 !ROOT HANDLE!
01A2 FFFF
01A4 FFFF
01A6 5D26
             LDL
                   KST. MM_HANDLE(R2), RR6
01A8 0000
             !SET ROOT ENTRY # IN G_AST !
                   KST.MM HANDLE[2](R2), #0
01AA 4D25
             LD
01AC 0004
01AE 0000
             ! SET ROOT CLASSIFICATION !
                   RR6, #SYSTEM_LOW
01B0 1406
             LDL
01B2 0000
01B4 0000
                   KST.CLASS(R2), RR6
01B6 5D26
             LDL
01B8 000A
             ISET MENTOR SEG #!
                  KST.M_SEG_NO(R2), #0
01BA 4C25
             LDB
01BC 000E
01BE 0000
             !INITIALIZE FREE KST ENTRIES
             FOR DEMO. NOT FULL KST!
                   R1, #10
01C0 2101
             LD
01C2 000A
             DO
                   R1, #0
             CP
01C4 0B01
01C6 0000
01C8 5E0E
              IF EQ THEN EXIT FI
01CA 01D0'
01CC 5E08
01CE 01DE
                   R2. #SIZEOF KST_REC
01D0 0102
              ADD
01D2 0010
```

```
0 1D4 4C25 LDB KST.M_SEG_NO(R2), #%FF
0 1D6 000E
0 1D8 FFFF
0 1DA AB10 DEC R1
0 1DC E8F3 OD
0 1DE 9E08 RET
0 1E0 END CREATE_KST
```

```
* EVENTCOUNT IS ADVANCED BY
           * INVOCATION OF MM_ADVANCE.
           * PROCESSES THAT ARE AWAITING
           * THIS EVENT OCCURRENCE ARE
           * REMOVED FROM THE BLOCKED LIST*
           * AND MADE READY.
                             THE READY
           * LISTS ARE THEN CHECKED TO
           * INSURE PROPER SHEDULING IS
           * EFFECTED.
                      IF NECESSARY VIR- *
           * TUAL PREEMPTS ARE SENT TO ALL*
           * THOSE VP'S BOUND TO LOWER
           * PRIORITY PROCESSES.
           *************
           * PARAMETERS:
             R1: HANDLE POINTER
             R2: INSTANCE (EVENT #)
           ***********
           * RETURNS:
             RO: SUCCESS CODE
           ************
           ENTRY
            ! ESTABLISH TEMPORARY VARIABLE
              STACK FRAME. !
                 R15, #SIZEOF TEMP
01E0 030F
            SUB
01E2 0012
            ! SAVE INPUT ARGUMENTS !
                  TEMP. HANDLE PTR(R15), R1
            LD
01E4 6FF1
01E6 0000
                  TEMP.EVENT_NR(R15), R2
01E8 6FF2
            LD
01EA 0002
            ! LOCK APT !
                 R4, APT.LOCK
01EC 7604
            LDA
01EE 0000°
01F0 5F00
            CALL K LOCK
01F2 0000*
            ! RETURNS WHEN APT IS LOCKED !
            ! ANNOUNCE EVENT OCCURRENCE BY
              INCREMENTING EVENTCOUNT IN G_AST!
            CALL MM_ADVANCE !R1:HANDLE PTR
01F4 5F00
01F6 0000*
                              R2: INSTANCE
                              RETURNS:
                              RO:SUCCESS CODE
                              RR 2: EVENTCOUNT!
                 RO, #SUCCEEDED
01F8 0B00
            CP
01FA 0002
            IF EQ THEN
01FC SEOE
01FE 0372'
             ! SAVE EVENTCOUNT !
            LDL TEMP.EVENT_VAL (R15), RR2
0200 5DF2
0202 0004
```

TC ADVANCE

PROCEDURE

01E0

```
! RESTORE INSTANCE !
0204 61F0
             LD
                  RO, TEMP.EVENT NR (R15)
0206 0002
             ! RESTORE HANDLE POINTER !
0208 61F1
             LD
                  R1, TEMP.HANDLE PTR(R15)
020A 0000
             ! SAVE HANDLE !
020C 5414
             LDL
                   RR4, HANDLE_VAL. HIGH (R1)
020E 0000
0210 5DF4
                   TEMP. HANDLE_HIGH (R15), RR4
             LDL
0212 000C
0214 6114
                   R4, HANDLE VAL.LOW (R1)
             LD
0216 0004
0218 6FF4
             LD
                   TEMP. HANDLE LOW (R15), R4
021A 0010
             ! AWAKEN ALL PROCESSES AWAITING
               THIS EVENT OCCURRENCE !
             ! GET FIRST BLOCKED PROCESS !
021C 6101
                   R1. APT.BLOCKED LIST
021E 000A
0220 7606
             LDA
                  R6, APT.BLOCKED_LIST
0222 000A'
            WAKE UP:
             DO
              ! DETERMINE IF AT END OF BLOCKED LIST !
0224 0B01
                   R1, #NIL
0226 FFFF
              IF EQ ! NO MORE BLOCKED PROCESSES !
0228 5E0E
              THEN EXIT FROM WAKE_UP
022A 0230
022C 5E08
022E 02B4*
              FI
              ! SAVE NEXT ITEM IN LIST !
0230 6117
              LD
                    R7, APT.AP.NEXT_AP(R1)
0232 0020
              ! DETERMINE IF PROCESS IS ASSOCIATED
                WITH CURRENT HANDLE !
0234 54F4
                   RR4, TEMP. HANDLE HIGH (R15)
              LPL
0236 000C
0238 5014
              CPL
                    RR4, APT. AP. HANDLE (R1)
023A 0030*
              IF EQ !HIGH HANDLE VALUE MATCHES!
023C 5E0E
              THEN
023E 02A2
0240 61F4
              LD R4, TEMP. HANDLE LOW (R15)
0242 0010
                    R4, APT.AP. HANDLE[2](R1)
0244 4514
             CP
0246 00341
              IF EQ ! HANDLE'S MATCH !
0248 5E0E
              THEN ! CHECK FOR INSTANCE MATCH !
024A 029C
024C 61F0
                      RO, TEMP. EVENT_NR (R15)
               LD
024E 0002
```

0250 0252	4B10 0036'	CP RO, APT. AP. INSTANCE (R1)
	5E0E 0296 •	IF EQ ! INSTANCE MATCHES ! THEN !DETERMINE IF THIS IS THE
0230	0290	OCCURRENCE THE PROCESS WAITING FOR !
	54F2 0004	LDL RR2, TEMP.EVENT_VAL (R 15)
	5012 0038'	CPL RR2, APT. AP. VALUE (R1)
	5E01 0290	IF GE !AWAITED EVENT HAS OCCURRED! THEN ! AWAKEN PROCESS !
		! REMOVE FROM BLOCKED LIST ! LD @R6, R7
0266	91F6	! SAVE LOCAL VARIABLES ! PUSHL @R15, RR6
	6112	SET LIST THREADING ARGUMENTS! LD R2, APT.AP.AFFINITY (R1)
026C		LDA R3, APT.READY_LIST(R2)
026E 0270		LDA R4, APT.AP.NEXT_AP
0274		LDA R5, APT.AP.PRI
0278		LDA R6, APT.AP.STATE
027C 027E	2107	LD R7, #READY
0280	5F00	LD R2, R1 CALL LIST_INSERT
0284	0000*	!R2: OBJ ID
		R3: LIST HEAD PTR R4: NEXT OBJ PTR
		R5: PRIORITY PTR
		R6: STATE PTR
		R7: STATE VALUE ! ! RESTORE LOCAL VARIABLES !
0286	95F6	POPL RR6, DR15
0288		LD R11, #REMOVED
028A 028C		ELSE !PROCESS STILL BLOCKED!
028E	0292	
0290	8DB8	CLR R11 FI ! END VALUE CHECK !
0292	5E08	ELSE ! PROCESS STILL BLOCKED!
	0298	CLR R11
0296	סממס	FI ! END INSTANCE CHECK !
	5E08	ELSE !PROCESS STILL BLOCKED!
029A	029E	

```
029C 8DB8
                CLR R11
              FI ! END HANDLE CHECK !
              ELSE ! PROCESS STILL BLOCKED!
029E 5E08
02A0 02A4 *
02A2 8DB8
               CLR R11
              FI ! END HIGH HANDLE CHECK !
              ! RESET AP POINTER REGISTERS !
                    R11, #REMOVED
              CP
02A4 0B0B
02A6 ABCD
              IF NE ! PROCESS IS STILL BLOCKED !
02A8 5E06
               THEN
02AA 02B0 °
02AC 7616
               LDA R6, APT.AP.NEXT_AP(R1)
02AE 00204
              FI
02B0 A171
              LD
                   R1, R7
02B2 E8B8
             OD
             ! DETERMINE IF ANY VIRTUAL PREEMPT
               INTERRUPTS ARE REQUIRED !
02B4 8D28
             CLR
                   R 2
            PREEMPT_CHECK:
             DO
                   R2, #NR_CPU * 2
02B6 0B02
              CP
02B8 0004
02BA 5EOE
             IF EQ !ALL READY LISTS CHECKED! THEN
02BC 02C2*
02BE 5E08
              EXIT FROM PREEMPT_CHECK
02C0 0366*
              FI
              ! CREATE PREEMPT VECTOR FOR VP'S !
02C2 8D18
              CLR
                    R 1
              DO ! FOR R1=1 TO NR_VP'S!
02C4 A910
               INC
                     R1
02C6 4B21
              CP
                     R1, APT. VP.NR VP (R2)
02C8 0010*
               IF GT ! PREEMPT VECTOR COMPLETED !
02CA 5E02
               THEN EXIT
02CC 02D2*
02CE 5E08
02D0 02D8 ·
               FI
02D2 ODF9
               PUSH DR15, #TRUE
02D4 0001
02D6 E8F6
              O D
              ! # TO PREEMPT !
02D8 9D38
              CLR
                    R 3
02DA 6124
              LD
                   R4, APT.VP.NR_VP(R2)
02DC 0010 ·
              ! # OF VP'S !
              ! GET FIRST READY PROCESS !
02DE 6121
              LD
                   R1. APT.READY LIST(R2)
02E0 0006 *
             CHECK_RDY_LIST:
              DO
```

```
! SEE IF READY LIST IS EMPTY !
02E2 0B01
                CP R1, #NIL
02E4 FFFF
                IF EQ !LIST IS EMPTY!
02E6 5E0E
                THEN EXIT FROM CHECK_RDY_LIST
02E8 02EE
02EA 5E08
02EC 0324
                FI
02EE 4D11
                CP
                      APT. AP. STATE (R1) . #RUNNING
02F0 002A
02F2 0000
                IF EQ !PROCESS IS RUNNING!
02F4 5E0E
                 THEN !DON'T PREEMPT IT!
02F6 030C
02F8 6115
                  LD
                        R5, APT.AP.VP_ID(R1)
02FA 002E
                  !COMPUTE LOCATION IN PREEMPT VECTOR!
02FC 4325
                  SUB
                        R5, APT. VP.FIRST (R2)
02FE 0014 1
0300 74F6
                  LDA
                        R6, R15 (R5)
0302 0500
0304 OD65
                        OR6, #FALSE
                 LD
0306 0000
0308 5E08
                 ELSE ! PREEMPT IT !
030A 030E
030C A930
                  INC
                       R3
                FI
030E AB40
                      R4
                DEC
0310 0B04
                CP
                      R4, #0
0312 0000
                IF EQ
                      ! ALL VP'S VERIFIED!
0314 SEOE
                THEN
0316 031C1
0318 5E08
                  EXIT FROM CHECK_RDY_LIST
031A 03241
                PI
                ! GET NEXT AP IN READY LIST !
031C 6110
               LD
                     RO, APT. AP. NEXT_AP (R1)
031E 0020'
0320 A101
                      R1, R0
               LD
0322 E8DF
               OD !END CHECK_RDY_LIST!
               ! SET NECESSARY PREEMPTS !
0324 6124
              LD
                     R4, APT.VP.NR_VP(R2)
0326 0010
0328 6121
                     R1. APT. VP. FIRST (R2)
              LD
032A 0014
             SEND_PREEMPT:
               DO
032C 97F0
                     RO, DR15
                POP
                ! CHECK TEMPLATE !
032E 0B00
               CP
                      RO, #TRUE
0330 0001
                IF EQ !CAN BE PREEMPTED!
```

```
THEN
0332 5E0E
0334 03501
0336 OB03
               CP R3, #0
0338 0000
               IF GT !PREEMPTS REQUIRED!
033A 5E02
                THEN !PREEMPT IT!
033C 0350'
                 !SAVE ARGUMENTS!
033E 93F1
                 PUSH DR15, R1
0340 91F2
                 PUSHL @R15, RR2
                 PUSH DR15, R4
0342 93F4
                 CALL SET_PREEMPT
0344 5F00
0346 0000*
                  !R1: VP ID!
                 ! RESTORE ARGUMENTS !
0348 97F4
                 POP R4, aR15
                 POPL RR2, @R15
034A 95F2
                 POP R1, aR15
034C 97F1
034E AB30
                 DEC R3
               PI
              FI
0350 A911
             INC R1, #2
0352 AB40
             DEC
                   R4
0354 0B04
             CP
                   R4. #0
0356 0000
             IF EQ !STACK RESTORED!
0358 5E0E
              THEN
035A 03601
035C 5E08
               EXIT
035E 0362
             FI
0360 E8E5
             OD !END SEND_PREEMPT!
             ! CHECK NEXT READY LIST !
0362 A921
            INC R2, #2
0364 E8A8
            OD ! END PREEMPT_CHECK!
            ! UNLOCK APT !
0366 7604
            LDA R4, APT.LOCK
0368 0000
036A 5F00
            CALL K UNLOCK
036C 0000*
            ! RESTORE SUCCESS CODE !
036E 2100
            LD RO, #SUCCEEDED
0370 0002
            FI
            ! RESTORE STACK !
0372 010F
           ADD R15, #SIZEOF TEMP
0374 0012
0376 9E08
           RET
0378
          END TC_ADVANCE
```

```
TC AWAIT
0378
                                 PROCEDURE
          * CHECKS USER SPECIFIED VALUE
           * AGAINST CURRENT EVENTCOUNT
           * VALUE. IF USER VALUE IS LESS *
           * THAN OR EQUAL EVENTCOUNT THEN*
           * CONTROL IS RETURNED TO USER. *
           * ELSE USER IS BLOCKED UNTIL
           * EVENT OCCURRENCE.
           **********
           * PARAMETERS:
              R1: HANDLE POINTER
             R2: INSTANCE (EVENT #)
               RR4: SPECIFIED VALUE
           **************
           * RETURNS:
              RO: SUCCESS CODE
           *************
           ENTRY
             : ESTABLISH STACK FRAME FOR
               TEMPORARY VARIABLES. !
                  R15, #SIZEOF TEMP
             SUB
0378 030F
037A 0012
             ! SAVE INPUT PARAMETERS !
                  TEMP. HANDLE_PTR (R15) , R1
037C 6FF1
             LD
037E 0000
                  TEMP.EVENT_NR (R15), R2
0380 6FF2
             LD
0382 0002
                   TEMP. EVENT VAL (R15), RR4
0384 5DF4
             LDL
0386 0004
             ! LOCK APT !
                   R4, APT.LOCK
0388 7604
             LDA
038A 0000°
038C 5F00
             CALL
                   K_LOCK
038E 0000*
             ! RETURNS WHEN APT IS LOCKED !
             ! GET CURRENT EVENTCOUNT !
                   MM READ EVENTCOUNT
             CALL
0390 5F00
0392 0000*
                   !R1:HANDLE POINTER
                    R2: INSTANCE
                   RETURNS:
                    RO: SUCCESS_CODE
                    RR4: EVENTCOUNT!
                   RO, #SUCCEEDED
0394 OB00
             CP
0396 0002
             IF EQ THEN
0398 5E0E
039A 0440
             ! DETERMINE IF REQUESTED EVENT
               HAS OCCURRED !
                   RR6, TEMP. EVENT_VAL (R15)
039C 54F6
             LDL
039E 0004
                   RR6, RR4
03A0 9046
             CPL
```

```
IF GT !EVENT HAS NOT OCCURRED!
03A2 5E02
              THEN !BLOCK PROCESS!
03A4 0440*
                ! IDENTIFY PROCESS !
               CALL RUNNING_VP !RETURNS:
03A6 5F00
03A8 0000*
                                   R1: VP ID
                                   R3:CPU #!
                ! SAVE RETURN VARIABLES !
03AA 6FF1
               LD
                      TEMP. ID_ VP (R 15) , R1
03AC 0008
03AE 6FF3
                     TEMP. CPU_NUM (R15), R3
               LD
03B0 000A
03B2 6118
               LD
                     R8, APT. RUNNING_LIST (R1)
03B4 0002*
                ! RESTORE REMAINING ARGUMENTS !
03B6 61F2
               LD
                     R2, TEMP.EVENT_NR(R15)
03B8 0002
03BA 61F1
               LD
                      R1. TEMP. HANDLE PTR (R15)
03BC 0000
               ! SAVE EVENT DATA !
03BE 5414
               LDL
                      RR4, HANDLE_VAL. HIGH (R1)
03C0 0000
03C2 5D84
                     APT.AP.HANDLE(R8), RR4
               LDL
03C4 0030*
03C6 6114
               LD
                     R4, HANDLE VAL.LOW (R1)
0308 0004
03CA 6F84
               LD
                     APT.AP.HANDLE[2](38), R4
03CC 00341
03CE 6F82
               LD
                     APT.AP.INSTANCE (R8), R2
03D0 0036 *
03D2 54F6
                     RR6, TEMP. EVENT VAL (R15)
               LDL
03D4 0004
03D6 5D86
               LDL
                      APT. AP. VALUE (R8) . RR6
03D8 0038°
               ? REMOVE PROCESS FROM READY LIST !
                     Ri. APT. AP. AFFINITY (R8)
03DA 6131
               LD
03DC 002C*
03DE 6112
                     R2, APT. READY LIST (R1)
               LD
03E0 0006'
               ! SEE IF PROCESS IS FIRST
                 ENTRY IN READY LIST !
03E2 8B62
                      R2, R8
               CP
               IF EQ !INSERT NEW READY LIST HEAD!
03E4 5E0E
03E6 03F4'
03E8 6183
                 LD
                        R3, APT.AP.NEXT_AP(E8)
03EA 0020'
03EC 6F13
                        APT.READY_LIST(R1), R3
                 LD
03EE 0006 *
03F0 5E08
                ELSE !DELETE FROM LIST BODY!
03F2 040E*
                 DO
03F4 6123
                  LD
                         R3, APT.AP.NEXT AP(R2)
```

```
03F6 0020"
03F8 8B83
                   CP
                       R3, R8
                   IF EQ !FOUND ITEM IN LIST!
03FA 5E0E
                    THEN
03FC 040A*
03FE 6183
                     LD
                            R3, APT.AP.NEXT_AP(R8)
0400 0020
0402 6F23
                     LD
                            APT. AP. NEXT_AP (R2), R3
0404 0020
0406 5E08
                     EXIT
0408 040E*
                   FI
040A A132
                   LD
                         R2. R3
040C E8F3
                  OD
                FI
                !THREAD PROCESS IN BLOCKED LIST!
040E A182
                LD
                      R2, R8
0410 7603
                LDA
                      R3, APT. BLOCKED_LIST
0412 000A .
0414 7604
                LDA
                      R4, APT. AP. NEXT_AP
0416 00201
0418 7605
                LDA
                      R5, APT. AP. PRI
041A 00281
                LDA
041C 7606
                     R6, APT. AP. STATE
041E 002A
0420 2107
                LD
                      R7, #BLOCKED
0422 0002
0424 5F00
                CALL
                      LIST INSERT !R2:OBJ ID
0426 0000*
                                    R3:LIST HEAD PTR
                                    R4: NEXT OBJ PTR
                                    R5: PRIORITY PTR
                                    R6:STATE PTR
                                    R7:STATE !
                ! GET CURRENT VP ID !
0428 61F1
                LD
                      R1, TEMP.ID_VP(R15)
042A 0008
042C 61F3
                LD
                      R3, TEMP.CPU_NUM(R15)
042E 000A
                ! SCHEDULE FIRST READY PROCESS !
                CALL TC_GETWORK
0430 5F00
                                  !R1:VP_ID
0432 0000
                                    R3:CPU #!
                ! UNLOCK APT !
0434 7604
                      R4, APT. LOCK
                LDA
0436 00001
0438 5F00
                      K_UNLOCK
                CALL
043A 0000*
                ! RESTORE SUCCESS CODE !
                    RO, #SUCCEEDED
043C 2100
                LD
043E 0002
             FI
            FI
             ! RESTORE STACK !
```

0440 010F ADD R15, #SIZEOF TEMP 0442 0012 0444 9E08 RET 0446 END TC_AWAIT

```
0446
          PROCESS_CLASS PROCEDURE
          * READS SECURITY ACCESS
          * CLASS OF CURRENT PROCESS *
          * IN APT. CALLED BY SEG
          * MGR AND EVENT MGR
          *********
          * LOCAL VARIABLES:
            R1: VP ID
          * R5: PROCESS ID
          *********
          * RETURNS:
             RR2: PROCESS SAC
          **********
          ENTRY
0446 7604
           LDA
                R4, APT. LOCK
0448 0000
044A 5F00
          CALL K LOCK
                        !R4:-APT.LOCK!
044C 0000*
044E 5F00
           CALL
                RUNNING_VP !RETURNS:
0450 0000*
                             R1: VP ID
                             R3: CPU #!
                 R5, APT. RUNNING_LIST(R1)
0452 6115
           LD
0454 00021
0456 5452
                 RR2.APT.AP.SAC(R5)
           LDL
0458 0024
            ! UNLOCK APT !
045A 7604
           LDA
                 R4, APT.LOCK
045C 0000"
045E 5F00
           CALL K_UNLOCK
0460 0000*
0462 9E08
           RET
           END PROCESS_CLASS
0464
```

```
0464
          GET DBR NUMBER
                                PROCEDURE
          * OBTAINS DBR NUMBER FROM APT
           * FOR THE CURRENT PROCESS.
           * CALLED BY SEGMENT MANAGER
          **********
           * LOCAL VARIABLES:
            R1: VP ID
           * R5: PROCESS ID
           ***********
          * RETURNS:
          * R1: DBR NUMBER
          ************
          ENTRY
            !NOTE: DBR # IS ONLY VALID WHILE PROCESS
            IS LOADED. THIS IS NO PROBLEM IN SASS
            AS ALL PROCESSES REMAIN LOADED. IN A
            MORE GENERAL CASE, THE DBR # COULD ONLY
            BE ASSUMED CORRECT WHILE THE APT IS LOCKED!
0464 7604
           LDA
                R4, APT. LOCK
0466 0000
0468 5F00
          CALL K LOCK !R4: -APT.LOCK!
046A 0000*
046C 5F00
           CALL RUNNING_VP ! RETURNS:
046E 0000*
                             R1: VP ID
                             R3:CPU #!
0470 6115
          LD
                 R5, APT. RUNNING_LIST(R1)
0472 0002
0474 6151
           LD
                 R1.APT.AP.DBR(R5)
0476 00221
           ! UNLOCK APT !
0478 7604
           LDA R4, APT.LOCK
047A 0000 ·
047C 5F00
           CALL K_UNLOCK
047E 0000*
0480 9E08
           RET
0482
          END GET_DBR_NUMBER
```

END TC

Appendix C

DISTRIBUTED MEMORY MANAGER LISTINGS

```
OBJ CODE
LOC
                  STMT SOURCE STATEMENT
        $LISTON $TTY
        DIST_MM MODULE
        CONSTANT
                            := 50
        CREATE_CODE
                            := 51
        DELETE CODE
        ACTIVATE_CODE
                            := 52
        DEACTIVATE_CODE
                            := 53
        SWAP_IN_CODE
                            := 54
        SWAP OUT CODE
                            := 55
                            := 2
        NR_CPU
                            := 54
        NR_KST_ENTRY
        MAX_SEG_SIZE
                            := 128
                            := 4
        MAX_DBR_NO
                            := 2
        KST_SEG_NO
                            := 10
        NR_OF_KSEGS
                            := 8
        BLOCK SIZE
                            := %F00
        MEM_AVAIL
                            := 10
        G_AST_LIMIT
                            := 1
        INSTANCE1
                            := 2
        INSTANCE2
                            := 95
        INVALID_INSTANCE
                             := 2
        SUCCEEDED
      TYPE
                        ARRAY [3
                                   WORD ]
        H ARRAY
                                    BYTE ]
                         ARRAY [16
        COM_MSG
                         WORD
        ADDRESS
                       RECORD
        G_AST_REC
                        LONG
         [UNIQUE_ID
          GLOBAL_ADDR
                       ADDRESS
          P L_ASTE_NO
                        WORD
                        WORD
          FLAG
                        WORD
          PAR_ASTE
          NR_ACTIVE
                        WORD
                        BYTE
          NO ACT_DEP
                        BYTE
          SIZE1
```

Z8000ASM 2.02

```
PG_TBL
                       ADDRESS
          ALIAS_TBL
                      ADDRESS
          SEQUENCER
                       LONG
          EVENT1
                       LONG
         EVENT2
                       LONG
                        WORD
        MM VP ID
                      ARRAY [ MAX_SEG_SIZE BYTE]
        SEG ARRAY
        $SECTION D_MM_DATA
       GLOBAL
0000
       MM CPU TBL ARRAY [NR_CPU MM_VP_ID]
       $SECTION AVAIL_MEM
       INTERNAL
        ! NOTE: MEM POOL IS LOCATED IN
         CPU LOCAL MEMORY. !
0000
                   ARRAY [MEM_AVAIL BYTE]
       MEM POOL
      GLOBAL
        ! NOTE: NEXT_BLOCK IS USED IN THE MM_ALLOCATE
         STUB AS AN OFFSET POINTER INTO THE BLOCK
         OF ALLOCATABLE MEMORY. IT IS INITIALIZED
         IN BOOTSTRAP LOADER. !
0F00
       NEXT BLOCK
                       WORD
        $SECTION MSG_FRAME_DCL
      INTERNAL
       !NOTE: THESE RECORDS ARE "OVERLAYS" OR "FRAMES" USED
        TO DEFINE MESSAGE FORMATS. NO MEMORY IS ALLOCATED !
       $ABS 0
       CREATE MSG RECORD [ CR_CODE
0000
                                              WORD
                                CE_MM_HANDLE H_ARRAY
                                CE_ENTRY_NO
                                              SHORT_INTEGER
                                CE FILL
                                              BYTE
                                CE SIZE
                                              WORD
                                CE_CLASS
                                              LONG]
       $ABS 0
0000
       DELETE MSG
                        RECORD [ DE_CODE
                                              WORD
                                DE_MM_HANDLE
                                              H ARRAY
                                DE ENTRY_NO
                                              SHORT INTEGER
                                              ARRAY[7 BYTE]]
                                DE_FILL
       $ABS 0
0000
       ACTIVATE_MSG RECORD [ ACT_CODE
                                              WORD
                                A DBR NO
                                              WORD
                                A_MM_HANDLE
                                             H ARRAY
                                A_ENTRY_NO
                                              SHORT_INTEGER
                                A_SEGMENT_NO SHORT_INTEGER
                                A FILL
                                              LONG
```

```
$ABS 0
0000
        DEACTIVATE_MSG RECORD[DEACT_CODE WORD
                                 D_DBR_NO
                                               WORD
                                 D_MM_HANDLE
                                              H_ARRAY
                                 D FILL
                                               ARRAY[ 3 WORD ]]
        $ABS 0
0000
        SWAP IN MSG
                         RECORD [S_IN_CODE
                                             WORD
                                 SI_MM_HANDLE
                                               H ARRAY
                                 SI DBR NO
                                               WORD
                                 SI_ACCESS_AUTH BYTE
                                 SI FILL1
                                               BYTE
                                               ARRAY[2 WORD]]
                                 SI_FILL
        $ABS 0
        SWAP_OUT_MSG
0000
                         RECORD [ S_OUT_CODE
                                              WORD
                                 SO_DBR_NO
                                               WORD
                                 SO_MM_HANDLE H_ARRAY
                                 SO_FILL
                                               ARRAY[3 WORD]]
        $ABS 0
        RET_SUC_CODE RECORD[SUC_CODE
0000
                                              BYTE
                                              ARRAY[ 15 BYTE]]
                                 SC_FILL
        $ABS 0
0000
                         RECORD [ R_SUC_CODE
                                               BYTE
        R_ACTIVATE_ARG
                                 R_FILL
                                               BYTE
                                 R_MM_HANDLE
                                               H ARRAY
                                               LONG
                                 R CLASS
                                               WORD
                                 R SIZE
                                 R FILL1
                                              WORD
        $ABS 0
0000
        MM HANDLE
                      RECORD
         [ ID
                    LONG
           ENTRY_NO WORD
          1
        EXTERNAL
                               WORD
               G_AST_LOCK
               G_AST ARRAY[G_AST_LIMIT G_AST_REC]
               K_LOCK
                                PROCEDURE
                                PROCEDURE
               K_UNLOCK
                                PROCEDURE
               GET_CPU_NO
                                PROCEDURE
               SIGNAL
                                PROCEDURE
               WAIT
```

GLOBAL \$SECTION D_MM_PROC

```
0000
            MM CREATE_ENTRY
                                   PROCEDURE
           * INTERFACE BETWEEN SEG MGR
             (CREATE_SEG PROCEDURE) AND
           * MMGR PROCESS (CREATE_ENTRY
            * PROCEDURE) . ARRANGES AND
           * PERFORMS IPC.
           *************
           * REGISTER USE:
            * PARAMETERS
              RO:SUCCESS_CODE (RET)
             R1:HPTR (INPUT)
              R2: ENTRY_NO (INPUT)
              R3:SIZE (INPUT)
             RR4:CLASS (INPUT)
           * LOCAL USE
              R6:MM_HANDLE ARRAY ENTRY
              R8: -COM_MSGBUF
              R13: - COM MSGBUF
           ************
            ENTRY
            !USE STACK FOR MESSAGE!
0000 030F
            SUB
                 R15, #SIZEOF COM_MSG
0002 0010
0004 A1FD
                  R13, R15
                           ! -COM_MSGBUF !
            LD
            !FILL COM_MSGBUF (LOAD MESSAGE). CREATE MSG
             PRAME IS BASED AT ADDRESS ZERO.
                                              IT IS
             OVERLAID ONTO COM_MSGBUF FRAME BY INDEXING
             EACH ENTRY (I.E. ADDING TO EACH ENTRY) THE
             BASE ADDRESS OF COM MSGBUF!
0006 4DD5
            LD
                  CREATE_MSG.CR_CODE (R13), #CREATE_CODE
0008 0000
000A 0032
000C 3116
            LD
                  R6,R1(#0) !INDEX TO MM HANDLE ENTRY!
000E 0000
0010 6FD6
                  CREATE_MSG.CE_MM_HANDLE[0](R13),R6
            LD
0012 0002
                  R6,R1(#2)
0014 3116
            LD
0016 0002
0018 6FD6
            LD
                  CREATE_MSG.CE_MM_HANDLE[ 1] (R13), R6
001A 0004
001C 3116
            LD
                  R6,R1(#4)
001E 0004
0020 6FD6
            LD
                  CREATE_MSG.CE_MM_HANDLE[2](R13),R6
0022 0006
0024 6FD2
            LD
                  CREATE_MSG.CE_ENTRY_NO (R13),R2
0026 0008
0028 5DD4
            LDL
                  CREATE_MSG.CE_CLASS(R13),RR4
002A 000C
```

```
002E 000A
0030 A1D8
             LD
                   R8,R13
0032 5F00
             CALL
                   PERFORM_IPC ! R8: ¬COM_MSGBUF!
0034 018C
             !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
                   RO
0036 8D08
             CLR
                   RLO, RET_SUC_CODE.SUC_CODE (R13)
0038 60D8
            LDB
0000 AE00
                   R15, #SIZEOF COM_MSG ! RESTORE STACK STATE!
003C 010F
             ADD
003E 0010
0040 9E08
             RET
         END MM_CREATE_ENTRY
0042
```

LD CREATE_MSG.CE_SIZE(R13),R3

002C 6FD3

```
0042
            MM DELETE_ENTRY
                                    PROCEDURE
           * INTERFACE BETWEEN SEG MGR
            * (DELETE SEG PROCEDURE) AND
            * MMGR (DELETE_ENTRY PROCEDURE) .*
            * ARRANGES AND PERFORMS IPC.
            ***********
            * REGISTER USE:
            * PARAMETERS
               RO:SUCCESS_CODE (RET)
               R1: HPTR (INPUT)
               R2: ENTRY_NO (INPUT)
            * LOCAL USE
               R6: MM HANDLE ARRAY ENTRY
               R8: -COM_MSGBUF
                                            *
               R13: - COM MSGBUF
            **************
            ENTRY
             !USE STACK FOR MESSAGE!
0042 030F
             SUB
                   R15, #SIZEOF COM MSG
0044 0010
0046 A1FD
                   R 13, R 15
                             ! ¬COM MSGBUF !
             LD
         !FILL COM_MSGBUF (LOAD MESSAGE). DELETE_MSG_FRAME
          IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
          COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
          ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUP!
0048 4DD5
                   DELETE_MSG.DE_CODE(R13), #DELETE CODE
             LD
004A 0000
004C 0033
004E 3116
             LD
                              !INDEX TO MM_HANDLE ENTRY!
                   R6,R1(#0)
0050 0000
0052 6FD6
             LD
                   DELETE_MSG.DE_MM_HANDLE[0](R13),R6
0054 0002
0056 3116
             LD
                   R6,R1(#2)
0058 0002
005A 6FD6
             LD
                   DELETE_MSG.DE_MM_HANDLE[ 1] (R13), R6
005C 0004
005E 3116
             LD
                   R6,R1(#4)
0060 0004
0062 6FD6
             LD
                   DELETE_MSG.DE_MM_HANDLE[2](R13),R6
0064 0006
0066 6FD2
             LD
                   DELETE_MSG.DE_ENTRY_NO(R13),R2
0068 0008
006A A1D8
             LD
                   R8,R13
006C 5F00
                   PERFORM_IPC ! R8: ¬COM_MSGBUF!
             CALL
006E 018C*
             !RETRIEVE SUCCESS CODE FROM RETURNED MESSAGE!
0070 8D08
             CLR
0072 60D8
             LDB
                   RLO, RET_SUC_CODE.SUC_CODE (R13)
0074 0000
0076 010F
             ADD
                   R15, #SIZEOF COM_MSG !RESTORE STACK STATE!
0078 0010
007A 9E08
             RET
007C
         END MM_DELETE ENTRY
```

```
007C
            MM ACTIVATE
                                    PROCEDURE
           * INTERFACE BETWEEN SEG MGR
            * (MAKE_KNOWN PROCEDURE) AND
            * MMGR
                    (ACTIVATE PROCEDURE) .
            * ARRANGES AND PERFORMS IPC.
            ************
            * REGISTER USE:
            * PARAMETERS
               R1: DBR_NO(INPUT)
               R2: HPTR (INPUT)
               R3: ENTRY NO
              R4:SEGMENT_NO
               R12:RET_HANDLE_PTR
            * LOCAL USE
               R8: -COM MSGBUF
               R13: - COM_MSGBUF
            * RETURNS:
               RO: SUCCESS CODE
               RR2:CLASS
               R4:SIZE
            **************
            ENTRY
             !USE STACK FOR MESSAGE!
007C 030F
                   R15, #SIZEOF COM_MSG
007E 0010
0080 A1FD
                  R 13, R 15
                            ! ¬COM MSGBUF !
             ! SAVE RETURN HANDLE POINTER !
0082 93FC
             PUSH
                  DR15. R12
         !FILL COM_MSGBUF (LOAD MESSAGE) . ACTIVATE_MSG FRAME
          IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
          COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
          ING TO EACH ENTRY) THE BASE ADDRESS OF COM MSGBUF!
0084 4DD5
             LD
                   ACTIVATE_MSG.ACT_CODE(R13), #ACTIVATE_CODE
0086 0000
0088 0034
008A 6FD1
             LD
                   ACTIVATE MSG.A DBR_NO(R13),R1
008C 0002
008E 3126
             LD
                  R6,R2(#0)
0090 0000
0092 6FD6
             LD
                   ACTIVATE MSG.A MM HANDLE[0](R13),R6
0094 0004
0096 3126
             LD
                   R6,R2(#2)
0098 0002
009A 6FD6
                   ACTIVATE_MSG.A_MM_HANDLE[ 1 ](R13),R6
            LD
009C 0006
009E 3126
                   R6,R2(#4)
            LD
00A0 0004
                   ACTIVATE_MSG.A_MM_HANDLE[2](R13),R6
00A2 6FD6
             LD
00A4 0008
                   ACTIVATE_MSG.A_ENTRY_NO(R13),RL3
00A6 6EDB
             LDB
0008 000A
```

```
LDB ACTIVATE_MSG.A_SEGMENT_NO(R13),RL4
OOAA 6EDC
00AC 000B
OOAE A1D8
                  R8,R13
             LD
00B0 5F00
             CALL PERFORM IPC ! (R8: -COM MSGBUF!
00B2 018C*
             ! RESTORE RETURN HANDLE POINTER !
00B4 97FC
             POP R12, 0R15
             ! UPDATE MM_HANDLE ENTRY !
                  RR6, R_ACTIVATE_ARG.R_MM_HANDLE (R13)
00B6 54D6
             LDL
00B8 0002
00BA 5DC6
             LDL
                  MM_HANDLE.ID (R12), RR6
00BC 0000
00BE 61D6
             LD
                  R6, R_ACTIVATE_ARG. R_MM_HANDLE[2](R13)
00C0 0006
00C2 6FC6
             LD
                   MM HANDLE. ENTRY NO (R12), R6
00C4 0004
             !RETRIEVE OTHER RETURN ARGUMENTS!
00C6 8D08
             CLR
00C8 60D8
             LDB
                   RLO, R_ACTIVATE_ARG.R_SUC_CODE (R13)
00CA 0000
00CC 54D2
             LDL RR2, R_ACTIVATE_ARG.R_CLASS (R13)
00CE 0008
00D0 61D4
             LD
                   R4, R_ACTIVATE_ARG. R_SIZE (R 13)
00D2 000C
00D4 010F
                  R15, #SIZEOF COM_MSG ! RESTORE STACK STATE!
             ADD
00D6 0010
00D8 9E08
             RET
OODA
        END MM_ACTIVATE
```

```
00DA
            MM_DEACTIVATE
                                     PROCEDURE
           · ********************************
            * INTERFACE BETWEEN SEG MGR
              (TERMINATE PROCEDURE) AND
            * MMGR (DEACTIVATE PROCEDURE) .
            * ARRANGES AND PERFORMS IPC.
            **************
            * REGISTER USE:
            * PARAMETERS
               RO: SUCCESS_CODE (RET)
               R1: DBR_NO(INPUT)
               R2: HPTR (INPUT)
            * LOCAL USE
               R6:MM HANDLE ARRAY ENTRY
               R8: -COM_MSGBUF
               R13: - COM MSGBUF
            ************
            ENTRY
             !USE STACK FOR MESSAGE!
00DA 030F
                   R15. #SIZEOF COM MSG
             SUB
00DC 0010
OODE A1FD
             LD
                   R13, R15
                              ! -COM MSGBUF !
         !FILL COM_MSGBUF (LOAD MESSAGE). DEACTIVATE_MSG FRAME
          IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
          COM MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
          ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
00E0 4DD5
                   DEACTIVATE MSG. DEACT CODE (R13),
             LD
                                     #DEACTIVATE_CODE
00E2 0000
00E4 0035
00E6 6FD1
                   DEACTIVATE_MSG.D_DBR_NO(R13),R1
             LD
00E8 0002
                   R6,R2(#0) !INDEX TO MM_HANDLE ENTRY!
00EA 3126
             LD
00EC 0000
                   DEACTIVATE MSG.D MM_HANDLE[0](R13), R6
OOEE 6FD6
             LD
00F0 0004
                   R6,R2(#2)
00F2 3126
             LD
00F4 0002
00F6 6FD6
             LD
                   DEACTIVATE MSG.D MM_HANDLE[ 1] (R13), R6
00F8 0006
00FA 3126
             LD
                   R6,R2(#4)
00FC 0004
OOFE 6FD6
             LD
                   DEACTIVATE_MSG.D_MM_HANDLE[2](R13),R6
0100 0008
0102 A1D8
             LD
                   R8,R13
                   PERFORM_IPC ! R8: -COM_MSGBUF!
0104 5F00
             CALL
0106 018C'
             IRETRIEVE SUCCESS CODE FROM RETURNED MESSAGE!
             CLR
                   R0
0108 8D08
010A 60D8
             LDB
                   RLO, RET_SUC_CODE.SUC_CODE (R13)
```

010C 0000 010E 010F ADD R15, #SIZEOF COM_MSG ! RESTORE STACK STATE! 0110 0010 0112 9E08 RET 0114 END MM_DEACTIVATE

```
0114
            MM SWAP IN
                                    PROCEDURE
           * INTERFACE BETWEEN SEG MGR (SM_*
             SWAP_IN PROCEDURE) AND MMGR
             (SWAP_IN PROCEDURE) . ARRANGES
            * AND PERFORMS IPC.
            ************
            * REGISTER USE:
            * PARAMETERS
               RO:SUCCESS_CODE (RET)
               R1: DBR_NO (INPUT)
               R2: HPTR (INPUT)
               R3: ACCESS
                             (INPUT)
            * LOCAL USE
               R6: MM HANDLE ARRAY ENTRY
               R8: -COM_MSGBUF
               R13: - COM MSGBUF
            *************
            ENTRY
             !USE STACK FOR MESSAGE!
0114 030F
                   R15. #SIZEOF COM MSG
             SUB
0116 0010
0118 A1FD
             LD
                   R13, R15
                             ! ¬COM_MSGBUF !
         !FILL COM_MSGBUF (LOAD MESSAGE). SWAP_IN_MSG FRAME
          IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
          COM MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
          ING TO EACH ENTRY) THE BASE ADDRESS OF COM MSGBUF!
011A 4DD5
                   SWAP_IN_MSG.S_IN_CODE(R13), #SWAP_IN_CODE
             LD
011C 0000
011E 0036
0120 3126
             LD
                              !INDEX TO MM HANDLE ENTRY!
                   R6.R2(#0)
0122 0000
0124 6FD6
                   SWAP_IN_MSG.SI_MM_HANDLE[0](R13),R6
             LD
0126 0002
0128 3126
                   R6,R2(#2)
             LD
012A 0002
012C 6FD6
                   SWAP_IN_MSG.SI_MM_HANDLE[1](R13),R6
             LD
012E 0004
0130 3126
             LD
                   R6,R2(#4)
0132 0004
                   SWAP IN MSG.SI_MM_HANDLE[2](R13),R6
0134 6FD6
             LD
0136 0006
0138 6FD1
             LD
                   SWAP_IN_MSG.SI_DBR_NO(R13),R1
013A 0008
                   SWAP IN MSG.SI ACCESS AUTH (R13), RL3
013C 6EDB
             LDB
013E 000A
0140 A1D8
             LD
                   R8,R13
                   PERFORM_IPC ! R8: -COM_MSGBUF!
0142 5F00
             CALL
0144 018C1
             RETRIEVE SUCCESS CODE FROM RETURNED MESSAGE!
0146 8D08
             CLR
                   RLO, RET SUC_CODE.SUC_CODE (R13)
0148 60D8
             LDB
```

014A 0000 014C 010F ADD R15, #SIZEOF COM_MSG !RESTORE STACK STATE! 014E 0010 0150 9E08 RET 0152 END MM_SWAP_IN

```
0152
            MM_SWAP_OUT
                                     PROCEDURE
           · ***********************
            * INTERFACE BETWEEN SEG MGR (SM_*
            * SWAP_OUT PROCEDURE) AND MMGR
            * (SWAP_OUT PROCEDURE). ARRANGES*
            * AND PERFORMS IPC.
            ************
            * REGISTER USE:
            * PARAMETERS
                                             *
               RO: SUCCESS_CODE (RET)
               R1: DBR_NO(INPUT)
              R2: HPTR (INPUT)
            * LOCAL USE
               R6: MM_HANDLE ARRAY ENTRY
               R8: ¬COM_MSGBUF
               R13: - COM MSGBUF
            ************
            ENTRY
             !USE STACK FOR MESSAGE!
0152 030F
             SUB
                  R15, #SIZEOF COM MSG
0154 0010
0156 A1FD
             LD
                   R13.R15
                           ! ¬COM_MSGBUF !
         !FILL COM_MSGBUF (LOAD MESSAGE). SWAP_OUT_MSG FRAME
          IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
          COM MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
          ING TO EACH ENTRY) THE BASE ADDRESS OF COM MSGBUF!
0158 4DD5
             LD
                   SWAP OUT MSG.S OUT CODE(R13), #SWAP OUT CODE
015A 0000
015C 0037
015E 3126
                   R6,R2(#0) !INDEX TO MM HANDLE ENTRY!
             LD
0160 0000
0162 6FD6
                   SWAP OUT MSG.SO MM HANDLE[0](R13), R6
             LD
0164 0004
                   R6,R2(#2)
0166 3126
             LD
0168 0002
016A 6FD6
             LD
                   SWAP_OUT_MSG.SO_MM_HANDLE[ 1](R13),R6
016C 0006
016E 3126
             LD
                   R6,R2(#4)
0170 0004
                   SWAP_OUT_MSG.SO_MM_HANDLE[2](R13),R6
0172 6FD6
             LD
0174 0008
0176 6FD1
             LD
                   SWAP OUT_MSG.SO_DBR_NO (R13),R1
0178 0002
017A A1D8
             LD
                   R8,R13
                   PERFORM IPC ! R8: -COM MSGBUF!
017C 5F00
             CALL
017E 018C'
             !RETRIEVE SUCCESS CODE FROM RETURNED MESSAGE!
0180 8D08
             CLR
                   RLO, RET_SUC_CODE.SUC_CODE (R13)
0182 60D8
             LDB
0184 0000
             ADD
                   R 15. #SIZEOF COM MSG ! RESTORE STACK STATE!
0186 010F
0188 0010
```

018A 9E08 RET 018C END MM_SWAP_OUT

```
· **********************
           * SERVICE ROUTINE TO ARRANGE AND
           * PERFORM IPC WITH THE MEM MGR PROC *
           ***************
           * REGISTER USE:
           * PARAMETERS
                                             *
             R8: ¬COM_MSG(INPUT)
           * LOCAL USE
                                             *
             R1, R2: WORK REGS
             R4: ¬G_AST_LOCK
           * R13: ¬COM MSGBUF
           ***************
           ENTRY
            PUSH
                  @R15,R13 !-COM_MSGBUF!
018C 93FD
018E 5F00
            CALL GET CPU_NO !RET-R1:CPU_NO!
0190 0000*
0192 A112
                  R2,R1
            LD
                  R1,MM_CPU_TBL(R2) !MM_VP_ID!
0194 6121
            LD
0196 0000
                 R4,G_AST_LOCK
0198 7604
           LDA
019A 0000*
019C 5F00
           CALL K LOCK
019E 0000*
           CALL SIGNAL !R1: MM VP ID, R8: -COM_MSGBUF!
01A0 5F00
01A2 0000*
            POP
                  R13, @R15
01A4 97FD
01A6 A1D8
                  R8,R13
                         !-COM_MSGBUF!
            LD
            PUSH @R15,B13
01A8 93FD
01AA 5F00
           CALL WAIT !R8: -COM_MSGBUF!
01AC 0000*
           LDA R4,G_AST_LOCK
01AE 7604
01B0 0000*
            CALL K_UNLOCK
01B2 5F00
01B4 0000*
01B6 97FD
                  R13, @R15
            POP
01B8 9E08
            RET
        END PERFORM_IPC
01BA
```

PROCEDURE

PERFORM_IPC

018C

```
01BA
           MM ALLOCATE
                         PROCEDURE
           * ALLOCATES BLOCKS OF CPU*
           * LOCAL MEMORY. EACH
           * BLOCK CONTAINS 256
           * BYTES OF MEMORY.
           *******
           * PARAMETERS:
              R3: # OF BLOCKS
             RETURNS:
             R2: STARTING ADDR
           * LOCAL:
              R4: BLOCK POINTER
           ************
           ENTRY
             ! NOTE: THIS PROCEDURE IS ONLY A STUB
              OF THE ORIGINALLY DESIGNED MEMORY
              ALLOCATING MECHANISM. IT IS USED
              BY THE PROCESS MANAGEMENT DEMONSTRATION
              TO ALLOCATE CPU LOCAL MEMORY FOR ALL
              MEMORY ALLOCATION REQUIREMENTS. IN AN
              ACTUAL SASS ENVIRONMENT, THIS WOULD
              BE BETTER SERVED TO HAVE SEPARATE
              ALLOCATION PROCEDURES FOR KERNEL AND
              SUPERVISOR NEEDS. (E.G., KERNEL_ALLOCATE
              AND SUPERVISOR ALLOCATE). !
            ! COMPUTE SIZE OF MEMORY REQUESTED !
                   R3, #BLOCK_SIZE
01BA B331
            SLL
01BC 0008
             ! COMPUTE OFFSET OF MEMORY THAT IS
              TO BE ALLOCATED !
01BE 6104
            LD
                   R4, NEXT_BLOCK !OFFSET!
01C0 0F00 .
01C2 7642
            LDA
                   R2, MEM POOL (R4) !START ADDR!
01C4 0000°
01C6 8134
            ADD
                  R4, R3 !UPDATE OFFSET!
            ! UPDATE OFFSET IN SECTION OF AVAILABLE
              MEMORY TO INDICATE THAT CURRENTLY
              REQUESTED MEMORY IS NOW ALLOCATED !
                   NEXT_BLOCK, R4 !SAVE OFFSET!
01C8 6F04
            LD
OICA OFOO!
01CC 9E08
            RET
```

END MM ALLOCATE

01CE

```
01CE
            MM TICKET
                               PROCEDURE
           · **********************
            * RETURNS CURRENT VALUE OF
            * SEGMENT SEQUENCER AND
            * INCREMENTS SEQUENCER VALUE*
            * FOR NEXT TICKET OPERATION *
            **********
            * PARAMETERS:
            * R1: SEG HANDLE PTR
            * RETURNS:
              RR4: TICKET VALUE
            * LOCAL VARIABLES:
              RR6: SEQUENCER VALUE
               R8: G AST ENTRY #
            ************
            ENTRY
             ! SAVE HANDLE PTR !
01CE 93F1
             PUSH
                    DR 15, R1
             ! LOCK G_AST !
01D0 7604
             LDA
                  R4, G_AST_LOCK
01D2 0000*
01D4 5F00
             CALL K_LOCK
01D6 0000*
             ! RESTORE HANDLE PTR !
01D8 97F1
             PO P
                   R1, DR15
             ! GET G_AST ENTRY # !
01DA 6118
                   R8, MM_HANDLE. ENTRY_NO (R1)
             LD
01DC 0004
             ! GET TICKET VALUE !
01DE 5486
             LDL
                   RR6, G_AST.SEQUENCER(R8)
01E0 0014*
             ! SET RETURN REGISTER VALUE !
01E2 9464
             LDL
                   RR4, RR6
             !ADVANCE SEQUENCER FOR NEXT
              TICKET OPERATION!
01E4 1606
             ADDL RR6, #1
01E6 0000
01E8 0001
             ! SAVE NEW SEQUENCER VALUE IN G_AST !
01EA 5D86
             LDL G AST. SEQUENCER (R8), RR6
01EC 0014*
             ! UNLOCK G AST !
             ! SAVE RETURN VALUES !
01EE 91F4
             PUSHL DR 15, RR4
01F0 7604
                  R4, G_AST_LOCK
             LDA
01F2 0000*
01F4 5F00
                 K UNLOCK
            CALL
01F6 0000*
             ! RETRIEVE RETURN VALUES !
01F8 95F4
            POPL RR4, DR 15
01FA 9E08
            RET
01FC
            END MM_TICKET
```

```
01FC
           MM READ EVENTCOUNT
                                PROCEDURE
           · ******************************
           * READS CURRENT VALUE OF THE
            * EVENTCOUNT SPECIFIED BY THE *
           * USER.
           ***********
            * PARAMETERS:
              R1: SEG HANDLE PTR
              R2: INSTANCE (EVENT #)
           ***********
           * RETURNS:
              RR4: EVENTCOUNT VALUE
           *********
           * LOCAL VARIABLES:
              RR6: SEQUENCER VALUE
                                         *
              R8: G AST ENTRY #
           **************
           ENTRY
            ! SAVE INPUT PARAMETERS !
01FC 93F1
                  DR15, R1
            PUSH
01FE 93F2
            PUSH
                  OR15, R2
            ! LOCK G_AST !
0200 7604
                  R4, G_AST_LOCK
            LDA
0202 0000*
0204 5F00
            CALL K_LOCK
0206 0000*
            ! RESTORE INPUT PARAMETERS !
0208 97F2
            PO P
                  R2. DR15
020A 97F1
            POP
                  R1, DR15
            ! GET G AST ENTRY # !
020C 6118
                  R8, MM_HANDLE. ENTRY_NO (R1)
            LD
020E 0004
            ! READ EVENTCOUNT !
            ! CHECK WHICH EVENT # !
            IF R2
0210 0B02
             CASE
                   #INSTANCE1 THEN
0212 0001
0214 5E0E
0216 0224
0218 5484
              LDL
                    RR4, G_AST.EVENT1(R8)
021A 0018*
021C 2100
              LD
                    RO, #SUCCEEDED
021E 0002
0220 SE08
             CASE
                   #INSTANCE2 THEN
0222 023C*
0224 OB02
0226 0002
0228 SEGE
022A 02381
022C 5484
              LDL
                    RR4, G AST. EVENT2(R8)
022E 001C*
0230 2100
              LD
                    RO, #SUCCEEDED
0232 0002
```

```
ELSE !INVALID INPUT!
0234 5E08
0236 023C*
0238 2100
             LD RO, #INVALID_INSTANCE
023A 005F
             FI
             ! NOTE: NO VALUE IS RETURNED IF
              USER SPECIFIED INVALID EVENT #!
             ! SAVE RETURN VALUES !
             PUSHL DR15, RR4
023C 91F4
             ! UNLOCK G_AST !
            LDA R4, G_AST_LOCK
023E 7604
0240 0000*
0242 5F00
            CALL K_UNLOCK
0244 0000*
            ! RESTORE RETURN VALUES !
0246 95F4
            POPL RR4, DR15
0248 9E08
            RET
            END MM_READ_EVENTCOUNT
024A
```

```
024A
           MM ADVANCE
                                    PROCEDURE
          ***************
           * DETERMINES G AST OFFSET FROM
           * SEGMENT HANDLE AND INCREMENTS
           * THE INSTANCE (EVENT #) SPECIFIED *
           * BY THE CALLER.
                            THIS IN EFFECT
           * ANNOUNCES THE OCCURRENCE OF THE *
           * EVENT. THE NEW VALUE OF THE
           * EVENTCOUNT IS RETURNED TO THE
           * CALLER.
           ***********
           * PARAMETERS:
              R1: HANDLE POINTER
              R2: INSTANCE (EVENT #)
                                            *
           **************
           * RETURNS:
              RR2: NEW EVENTCOUNT VALUE
           *************
           ENTRY
            ! SAVE INPUT PARAMETERS !
024A 93F1
            PUSH DR15, R1
024C 93F2
            PUSH DR15, R2
            ! LOCK G AST !
024E 7604
                 R4, G_AST_LOCK
            LDA
0250 0000*
0252 5F00
            CALL K LOCK
0254 0000*
            ! RESTORE INPUT PARAMETERS !
0256 97F2
            POP
                  R2, aR15
0258 97F1
            PO P
                  R1, DR15
            ! GET G_AST OFFSET !
025A 6114
                  R4, MM_HANDLE.ENTRY_NO (R1)
025C 0004
            ! DETERMINE INSTANCE !
            IF R2
025E 0B02
            CASE #INSTANCE1 THEN
0260 0001
0262 5E0E
0264 027C*
0266 5442
                    RR2, G AST.EVENT1(R4)
              LDL
0268 0018*
026A 1602
                    RR2, #1
              ADDL
026C 0000
026E 0001
              ! SAVE NEW EVENTCOUNT !
0270 5D42
              LDL
                   G_AST.EVENT1 (R4), RR2
0272 0018*
0274 2100
             LD
                  RO, #SUCCEEDED
0276 0002
0278 5E08
            CASE #INSTANCE2 THEN
027A 029E
027C 0B02
027E 0002
```

```
0280 5E0E
0282 029A'
0284 5442
                    RR2, G_AST.EVENT2(R4)
               LDL
0286 001C*
0288 1602
              ADDL RR2, #1
028A 0000
028C 0001
               ! SAVE NEW EVENTCOUNT !
028E 5D42
              LDL G_AST.EVENT2 (R4), RR2
0290 001C*
0292 2100
              LD RO, #SUCCEEDED
0294 0002
0296 5E08
             ELSE !INVALID INPUT!
0298 029E
029A 2100
                    RO, #INVALID_INSTANCE
               LD
029C 005F
             ΡI
             ! NOTE: AN INVALID INSTANCE VALUE
               WILL NOT AFFECT EVENT DATA !
             ! UNLOCK G_AST !
             LDA R4, G_AST_LOCK
029E 7604
02A0 0000*
02A2 5F00
            CALL K_UNLOCK
02A4 0000*
02A6 9E08
            RET
02A8
           END MM_ADVANCE
           END DIST MM
```

Appendix D

GATE KEEPER LISTINGS

```
Z8000ASM 2.02
LOC OBJ CODE STMT SOURCE STATEMENT
       KERNEL_GATE_KEEPER MODULE
       $LISTON $TTY
       CONSTANT
                              := 1
:= 2
         ADVANCE_CALL
AWAIT_CALL
         CREATE_SEG_CALL
                              := 3
         DELETE_SEG_CALL
MAKE_KNOWN_CALL
                              := 4
                              := 5
                              := 6
         READ_CALL
         SM_SWAP_IN_CALL
SM_SWAP_OUT_CALL
                              := 7
                              := 8
                              := 9
         TERMINATE_CALL
         TICKET_CALL
                              := 10
                              := 11
         WRITE CALL
         WRITELN_CALL
                              := 12
                              := 13
         CRLF_CALL
                              := %OFC8 !PRINT CHAR!
         WRITE
         WRITELN
                              := %OFCO !PRINT MSG!
         CRLF
                              := %OFD4 !CAR RET/LINE FEED!
                               := %A902
         MONITOR
                              := 32
         REGISTER_BLOCK
         TRAP_CODE_OFFSET := 36
         INVALID_KERNEL_ENTRY := %BAD
       GLOBAL
         GATE_KEEPER_ENTRY LABEL
       EXTERNAL
                              PROCEDURE
         ADVANCE
                              PROCEDURE
         AWAIT
         CREATE SEG
                              PROCEDURE
         DELETE_SEG
                               PROCEDURE
         MAKE KNOWN
                              PROCEDURE
         READ
                               PROCEDURE
         SM_SWAP_IN
                               PROCEDURE
         SM_SWAP_OUT
                              PROCEDURE
```

PROCEDURE

PROCEDURE

LABEL

TERMINATE

KERNEL_EXIT

TICKET

\$SECTION KERNEL_GATE_PROC 0000 GATE_KEEPER_MAIN PROCEDURE ENTRY GATE_KEEPER_ENTRY: ! SAVE REGISTERS ! 0000 030F R15, #REGISTER_BLOCK SUB 0002 0020 0004 1CF9 LDM @R15, R1, #16 0006 010F ! SAVE NSP ! 0008 93F2 PUSH @R15, R2 000A 7D27 LDCTL R2, NSP ! RESTORE INPUT REGISTERS ! 000C 2DF2 EX R2, DR15 ! SAVE REGISTER 2 ! 000E 93F2 PUSH DR15, R2 ! GET SYSTEM TRAP CODE ! 00 10 31F2 LD R2, R15 (#TRAP_CODE_OFFSET) 0012 0024 ! REMOVE SYSTEM CALL IDENTIFIER FROM SYSTEM TRAP INSTRUCTION ! 0014 8C28 CLRB RH2 ! NOTE: THIS LEAVES THE USER VISIBLE EXTENDED INSTRUCTION NUMBER IN R2 ! ! DECODE AND EXECUTE EXTENDED INSTRUCTION ! IF R2 ! NOTE: THE INITIAL VALUE FOR REGISTER 2 WILL BE RESTORED WHEN THE APPROPRIATE CONDITION IS FOUND ! 0016 0B02 CASE #ADVANCE CALL THEN 0018 0001 001A 5E0E 001C 0028 001E 97F2 POP R2, DR15 0020 5F00 CALL ADVANCE 0022 0000* 0024 5E08 CASE #AWAIT CALL THEN 0026 010C* 0028 9B02 002A 0002 002C 5E0E 002E 003A 0030 97F2 POP R2, DR15 CALL AWAIT 0032 5F00 0034 0000* 0036 SE08 CASE #CREATE_SEG_CALL THEN 0038 010C' 003A 0B02 003C 0003

INTERNAL

003E 5E0E

```
0040 004C*
0042 97F2
              POP R2, DR15
0044 5F00
              CALL CREATE SEG
0046 0000*
0048 5E08
            CASE #DELETE_SEG_CALL THEN
004A 010C*
004C 0B02
004E 0004
0050 SEOE
0052 005E
0054 97F2
              POP
                    R2, @R15
0056 5F00
              CALL DELETE_SEG
0058 0000*
005A 5E08
          CASE #MAKE_KNOWN_CALL THEN
005C 010C*
005E 0B02
0060 0005
0062 5E0E
0064 0070
0066 97F2
              POP R2, aR15
0068 5F00
              CALL MAKE_KNOWN
006A 0000*
006C 5E08
            CASE #READ CALL THEN
006E 010C*
0070 0B02
0072 0006
0074 5E0E
0076 00821
              POP R2, DR15
0078 97F2
007A 5F00
               CALL READ
007C 0000*
007E 5E08
            CASE #SM_SWAP_IN_CALL THEN
0080 010C*
C082 0B02
0084 0007
0086 5E0E
0088 0094*
008A 97F2
              POP R2, DR15
008C 5F00
               CALL SM_SWAP_IN
C08E 0000*
0090 5E08
            CASE #SM_SWAP_OUT_CALL THEN
0092 010C*
0094 0B02
0096 0008
0098 5E0E
009A 00A6 .
009C 97F2
              POP R2, DR15
009E 5F00
              CALL SM_SWAP_OUT
00A0 0000*
00A2 5E08
            CASE #TERMINATE_CALL THEN
00A4 010C*
00A6 0B02
00A8 0009
OUAA SEOE
```

```
00AC 00B8*
00AE 97F2
               POP R2, aR15
00B0 5F00
               CALL TERMINATE
00B2 0000*
00B4 5E08
             CASE #TICKET_CALL THEN
00B6 010C1
00B8 0B02
00BA 000A
OOBC SEOE
OOBE OOCA
00C0 97F2
               POP R2, aR15
00C2 5F00
                CALL TICKET
00C4 0000*
00C6 5E08
             CASE #WRITE_CALL THEN
00C8 010C
00CA 0B02
00CC 000B
OOCE SEGE
00D0 00DC*
00D2 97F2
                POP
                      R2, @R15
00D4 5F00
                CALL WRITE
00D6 0FC8
00D8 5E08
           CASE #WRITELN_CALL THEN
00DA 010C*
00DC 0B02
00DE 000C
00E0 5E0E
00E2 00EE*
00E4 97F2
               POP R2, aR15
00E6 5F00
               CALL WRITELN
OOE8 OFCO
00EA 5E08
             CASE #CRLF_CALL THEN
00EC 010C*
00EE 0B02
00F0 000D
00F2 5E0E
00F4 01001
00F6 97F2
               POP R2, aR15
00F8 5F00
               CALL
                     CRLF
OOFA OFD4
00FC 5E08
             ELSE ! INVALID KERNEL INVOCATION!
00FE 010C*
                ! RETURN TO MONITOR !
                ! NOTE: THIS RETURN TO MONITOR IS
                  FOR STUB USE ONLY. AN INVALID
                  KERNEL INVOCATION WOULD NORMALLY
                  RETURN TO USER. !
0100 7601
                     R1, $
               LDA
0102 01001
0104 2100
              LD RO, #INVALID_KERNEL_ENTRY
0106 OBAD
0108 5F00
             CALL MONITOR
010A A902
```

```
! SAVE REGISTERS ON KERNEL STACK !
             ! SAVE R1 !
             PUSH DR15, R1
010C 93F1
             ! GET ADDRESS OF REGISTER BLOCK !
010E 34F1
            LDA R1, R15 (#4)
0110 0004
             ! SAVE REGISTERS IN REGISTER BLOCK
              ON KERNEL STACK. !
0112 1019
             LDM @R1, R1, #16
0114 010F
             ! RESTORE R1 BUT MAINTAIN ADDRESS
              OF REGISTER BLOCK !
             EX R1, DR15
0116 2DF1
             ! SAVE R1 ON STACK !
0118 33F1
            LD R 15 (#4), R1
011A 0004
             ! RESTORE REGISTER BLOCK ADDRESS !
011C 97F1
            POP R1, DR15
             ! SAVE VALID EXIT SP VALUE !
011E 33F1
            LD R15 (#30), R1
0120 001E
             ! EXIT KERNEL BY MEANS OF HARDWARE
              PREEMPT HANDLER !
0122 5E08
          JP KERNEL_EXIT
0124 0000*
0126
          END GATE KEEPER MAIN
         END KERNEL_GATE_KEEPER
```

```
Z8000ASM 2.02
LOC OBJ CODE STMT SOURCE STATEMENT
       USER_GATE MODULE
       $LISTON $TTY
       CONSTANT
        ADVANCE_CALL
                         := 1
:= 2
        AWAIT CALL
        CREATE_SEG_CALL
                          := 3
        DELETE_SEG_CALL
        MAKE_KNOWN_CALL
                           := 5
        READ_CALL
                           := 6
        SM_SWAP_IN_CALL
        SM_SWAP_OUT_CALL
                           := 8
        TERMINATE_CALL
                           := 9
        TICKET CALL
                           ; = 10
        WRITE_CALL
                           := 11
        WRITELN_CALL
                           := 12
        CRLF CALL
                           := 13
       GLOBAL
       $SECTION USER_GATE_PROC
         ADVANCE
0000
                      PROCEDURE
        · ********************
         * PARAMETERS:
         * R1:SEGMENT #
         * R2:INSTANCE (ENTRY#) *
         ********
         * RETURNS:
         * RO:SUCCESS CODE *
         *********
         ENTRY
0000 7F01
         SC
RET
               #ADVANCE_CALL
0002 9E08
0004
        END ADVANCE
                      PROCEDURE
0004
        TIAWA
        · **********************
         * PARAMETERS:
         * R1:SEGMENT #
            R2: INSTANCE
            RR4:SPECIFIED VALUE *
         *********
         * RETURNS:
         * RO:SUCCESS CODE
         ********
         ENTRY
         SC #AWAIT_CALL
0004 7F02
0006 9E08
          RET
8000
        END AWAIT
```

```
CREATE_SEG PROCEDURE
0008
         · ********************
          * PARAMETERS:
            R1:MENTOR_SEG_NO
            R2: ENTRY NO
            R3:SIZE
            RR4: CLASS
          ********
          * RETURNS:
           RO:SUCCESS CODE
          *********
          ENTRY
0008 7F03
           SC
                 #CREATE_SEG_CALL
000A 9E08
           RET
000C
          END CREATE_SEG
000C
          DELETE_SEG PROCEDURE
         · **********************
          * PARAMETERS:
            R1: MENTOR_SEG_NO
            R2: ENTRY NO
          *********
          * RETURNS:
          * RO:SUCCESS CODE
          ***********
          ENTRY
000C 7F04
           SC
                #DELETE_SEG_CALL
000E 9E08
           RET
0010
          END DELETE_SEG
0010
          MAKE_KNOWN PROCEDURE
         · **********************
          * PARAMETERS:
            R1: MENTOR_SEG_NO
            R2:ENTRY_NO
            R3: ACCESS DESIRED
          *********
          * RETURNS:
           RO:SUCCESS CODE
          * R1:SEGMENT #
            R2: ACCESS ALLOWED
         *********
          ENTRY
0010 7F05
           SC
                 #MAKE_KNOWN_CALL
0012 9E08
           RET
0014
          END MAKE_KNOWN
0014
         READ
                       PROCEDURE
         · ********************
         * PARAMETERS:
           R1: SEGMENT #
         * R2:INSTANCE
          *************
```

```
* RETURNS:
           RO: SUCCESS CODE
                             380
            RR4: EVENTCOUNT
         ********
         ENTRY
0014 7F06
           SC
              #READ_CALL
0016 9E08
           RET
0018
         END READ
0018
         SM_SWAP_IN PROCEDURE
        * PARAMETERS:
           R1:SEGMENT #
         *********
         * RETURNS:
            RO: SUCCESS CODE
         ****************
         ENTRY
0018 7F07
           SC
                #SM_SWAP_IN_CALL
001A 9E08
          RET
001C
         END SM_SWAP_IN
001C
         SM SWAP OUT PROCEDURE
        · ***********************
         * PARAMETERS:
         * R1:SEGMENT #
         ********
                             skr
         * RETURNS:
            RO: SUCCESS CODE
         *********
         ENTRY
001C 7F08
                #SM_SWAP_OUT_CALL
           SC
001E 9E08
           RET
0020
         END SM_SWAP_OUT
0020
         TERMINATE
                      PROCEDURE
        · *******************
         * PARAMETERS:
            R1:SEGMENT #
         ********
         * RETURNS:
                             *
                             *
            RO: SUCCESS CODE
         *********
         ENTRY
0020 7F09
          SC
                #TERMINATE CALL
0022 9E08
           RET
         END TERMINATE
0024
                      PROCEDURE
0024
         TICKET
        ******************
         * PARAMETERS:
            R1: SEGMENT #
         ********
         * RETURNS:
```

0026 9E08	* R0:SUCCESS C * RR4:TICKET V ********** ENTRY SC #TICKET RET END TICKET	'ALUE * ***********
0028 7F0B 002A 9E08	ENTRY SC #WRITE_	PROCEDURE
002C 7F0C 002E 9E08	WRITELN ENTRY SC #WRITEL RET END WRITELN	
0030 7F0D 0032 9E08	CRLF ENTRY SC #CRLF_C RET END CRLF	PHOCEDURE

Appendix E

BOOTSTRAP_LOADER LISTINGS

Z8000ASM 2.02

LOC OBJ CODE STMT SOURCE STATEMENT

BOOTSTRAP_LOADER MODULE

\$LISTON \$TTY CONSTANT

TYPE

```
! ***** SYSTEM PARAMETERS ******* !
NR_CPU
              := 2
               := NR_CPU*4
NR_VP
NR AVAIL VP
              := NR CPU*2
MAX_DBR_NR
              := 10
STACK SEG
               := 1
STACK_SEG_SIZE := %100
STACK_BLOCK := STACK_$EG_SIZE/256
  ! * * OFFSETS IN STACK SEG * *!
STACK_BASE := STACK_SEG_SIZE-%10
STATUS REG_BLOCK: = STACK_SEG_SIZE-%10
INTERRUPT FRAME := STACK BASE-4
INTERRUPT_REG := INTERRUPT_FRAME-34
               := INTERRUPT_REG-2
N_S_P
               := STACK SEG SIZE-%E
F_C_W
! ***** SYSTEM CONSTANTS ***** !
ON
               := %FFFF
               := 0
OFF
              := 1
READY
              := %FFFF
NIL
INVALID
               := %EEEE
              := %5000
KERNEL FCW
AVAILABLE
              := 0
              := %FF
ALLOCATED
               := 12
SC OFFSET
MESSAGE ARRAY [16 BYTE]
ADDRESS WORD
MM VP ID WORD
VP INDEX
               INTEGER
MSG_INDEX INTEGER
```

```
MSG TABLE RECORD
     [ MSG
                    MESSAGE
       SENDER
                    VP_INDEX
       NEXT MSG
                    MSG_INDEX
                    ARRAY [6, WORD]
       FILLER
    VP TABLE RECORD
       DBR ADDRESS
       PRI
                     WORD
       STATE
                     WORD
       IDLE FLAG
                     WORD
       PREEMPT
                     WORD
       PHYS PROCESSOR WORD
       NEXT_READY_VP VP_INDEX
       MSG_LIST
                     MSG_INDEX
                     WORD
       EXT ID
       FILLER_1
                     ARRAY[ 7, WORD]
    1
EXTERNAL
    GET_DBR_ADDR
                    PROCEDURE
    CREATE STACK PROCEDURE
    LIST INSERT
                    PROCEDURE
    ALLOCATE_MMU
                   PROCEDURE
    UPDATE MMU IMAGE PROCEDURE
    MM ALLOCATE
                   PROCEDURE
    MM_ENTRY
                    LABEL
   IDLE ENTRY
                    LABEL
    PREEMPT_RET
                   LABEL
    BOOTSTRAP_ENTRY LABEL
    GATE_KEEPER_ENTRY LABEL
    NEXT_BLOCK
                    WORD
    MM_CPU_TBL ARRAY[NR_CPU MM_VP_ID]
   VPT
            RECORD
   LOCK
                   WORD
     RUNNING_LIST ARRAY[NR_CPU WORD]
     READY_LIST
                   ARRAY[ NR_CPU WORD]
     FREE_LIST
                   MSG_INDEX
     VIRT_INT_VEC ARRAY[ 1. ADDRESS ]
     FILLER_2
                   WORD
     VP.
                ARRAY [NR_VP, VP_TABLE]
     MSG_Q
                   ARRAY [NR_VP, MSG_TABLE]
    1
```

EXT_VP_LIST ARRAY[NR_AVAIL_VP WORD]
NEXT_AVAIL_MMU ARRAY[MAX_DBR_NR BYTE]

PRDS RECORD

[PHYS_CPU_ID WORD
LOG_CPU_ID INTEGER
VP_NR WORD
IDLE_VP VP_INDEX]

INTERNAL \$SECTION LOADER_DATA

! NOTE: THESE DECLARATIONS WILL NOT WORK IN A TRUE MULTIPROCESSOR ENVIRONMENT AS THEY ARE SUBJECT TO A "CALL." THEY MUST BE DECLARED AS A SHARED GLOBAL DATABASE WITH "RACE" PROTECTION (E.G., LOCK).!

0000 NEXT_AVAIL_VP INTEGER 0002 NEXT_EXT_VP INTEGER

\$SECTION LOADER_INT INTERNAL

	THIE	DNAL
0000		BOOTSTRAP PROCEDURE
		! **************************
		* CREATES KERNEL PROCESSES AND *
		* INITIALIZES KERNEL DATABASES.*
		* INCLUDES INITIALIZATION OF *
		* VIRTUAL PROCESSOR TABLE, *
		* EXTERNAL VP LIST, AND MMU *
		·
		* IMAGES. ALLOCATES MMU IMAGE *
		* AND CREATES KERNEL DOMAIN *
		* STACK FOR KERNEL PROCESSES. *

		* * * * * * * * * * * * * * * * * * *
		ENTRY
		! INITIALIZE PRDS AND MMU POINTER !
		! NOTE: THE FOLLOWING CONSTANTS ARE
		ONLY TO BE INITIALIZED ONCE. THIS
		WILL OCCUR DURING SYSTEM INITIALIZATION!
0000	4D05	LD PRDS. PHYS_CPU_ID, #%FFFF
	0000*	
0004	FFFF	
		! NOTE: LOGICAL CPU NUMBERS ARE ASSIGNED
		IN INCREMENTS OF 2 TO FACILITATE INDEXING
		(OFFSETS) INTO LISTS SUBSCRIPTED BY
		LOGICAL CPU NUMBER. !
0006	4D05	LD PRDS.LOG_CPU_ID, #2
0008	0002*	
0004	0002	
0001	0002	! SPECIFY NUMBER OF VIRTUAL PROCESSORS
		ASSOCIATED WITH PHYSICAL CPU. !
000C	4D05	LD PRDS. VP NR, #2
000E	0004*	- ·
0010		
		415 VIII 3144
	4D08	CLR NEXT_BLOCK
0014	0000*	
0016	4D08	CLR NEXT_AVAIL_VP
	0000	
		OTD VEWS BUS UP
	4D08	CLR NEXT_EXT_VP
001C	0002	
		! ESTABLISH GATE KEEPER AS SYSTEM CALL
		TRAP HANDLER !
		! GET BASE OF PROGRAM STATUS AREA!
0045	75.46	
OOIE	7D15	LDCTL R1, PSAP
		! ADD SYSTEM CALL OFFSET TO PSA BASE ADDR!
0020	0101	ADD R1, #SC_OFFSET
		WIN MRC OFFREI
0022	0000	
		! STORE KERNEL PCW IN PSA !
0024	0D15	LD @R1, #KERNEL_FCW
0026		
0020	2000	L CHORD IDDDDGG OF GIFT WHITE THE PROPERTY
		! STORE ADDRESS OF GATE KEEPER IN PROGRAM
		STATUS AREA AS SYSTEM TRAP HANDLER!
0028	A911	INC R1, #2
_		

```
002A 0D15
              LD
                        DR1, #GATE_KEEPER_ENTRY
002C 0000*
002E 8D18
              CLR
                        R1 ! NEXT_AVAIL_MMU INDEX !
               ! INITIALIZE ALL MMU IMAGES AS AVAILABLE !
           SET_MMU_MAP:
               DO
0030 4C15
                LDB
                        NEXT_AVAIL_MMU(R1), #AVAILABLE
0032 0000*
0034 0000
0036 A910
                INC
                        R1, #1
                ! CHECK FOR END OF TABLE !
0038 0B01
                CP
                        R1, #MAX DBR NR
003A 000A
003C 5E0E
               IF EQ THEN EXIT FROM SET_MMU_MAP FI
003E 0044
0040 5E08
0042 0046
0044 E8F5
              OD
              ! CREATE MEMORY MANAGER PROCESS !
0046 2103
              LD
                       R3, #STACK_BLOCK
0048 0001
              ! ALLOCATE AND INITIALIZE KERNEL
                DOMAIN STACK SEGMENT !
004A 5F00
              CALL
                       MM ALLOCATE !R3: # OF BLOCKS
004C 0000*
                                        RETURNS
                                        R2: START ADDR!
004E A121
              LD
                        R1, R2
0050 2103
              LD
                        R3, #KERNEL_FCW
0052 5000
0054 7604
                       R4, HH_ENTRY
              LDA
0056 0000*
0058 6105
              LD
                       R5. %FFFF !NSP!
005A FFFF
005C 7606
                     R6, PREEMPT_RET
              LDA
005E 0000*
0060 93F1
              PUSH
                       OR 15, R1 !SAVE STACK ADDR!
0062 030F
              SUB
                        R15, #8
0064 0008
0066 1CF9
                      @R15, R3, #4
              LDM
0068 0303
                       RO, R15
006A A1F0
             LD
              ! NOTE: ARGLIST FOR CREATE_STACK INCLUDES
                KERNEL_FCW, INITIAL IC, NSP, AND INITIAL
                RETURN POINT. !
006C 5F00
              CALL
                        CREATE_STACK ! (RO: ARGUMENT PTR
006E 0000*
                                          R1: TOP OF STACK
                                          R2-R14: INITIAL
                                          REG.STATES !
```

```
R15, #8 !OVERLAY ARGUMENTS!
0070 010F
            ADD
0072 0008
              ! ALLOCATE MMU IMAGE !
             CALL ALLOCATE_MMU
0074 5F00
                                        ! RETURNS:
0076 0000*
                                         (RO: DBR #) !
0078 2101
                       R1, #STACK SEG
                                         ! SEGMENT NO. !
             LD
007A 0001
007C 97F2
             POP
                       R2. DR15 !GET STACK ADDR!
007E 2103
                       R3, #0 ! WRITE ATTRIBUTE !
             L D
0000 0800
              ! SPECIFY NUMBER OF BLOCKS. COUNT STARTS
               FROM ZERO. (I.E., 1 BLOCK=0, 2=1, ETC.)!
                       R4, #STACK_BLOCK-1
0082 2104
              LD
0084 0000
              ! SAVE DBR # !
0086 93F0
                        DR 15. RO
             PUSH
              ! CREATE MMU ENTRY FOR MM STACK SEGMENT !
0088 5F00
                       UPDATE_MMU_IMAGE ! (RO: DBR #
              CALL
*0000 A800
                                          R1: SEGMENT #
                                          R2: SEG ADDRESS
                                          R3: SEG ATTRIBUTES
                                          R4: SEG LIMITS) !
             ! RESTORE DBR # !
008C 97F0
             POP
                         RO. DR15
              ! GET ADDRESS OF MMU IMAGE !
008E 5F00
             CALL
                       GET DBR ADDR ! (RO: DBR #)
0090 0000*
                                         RETURNS:
                                         (R1: DBR ADDRESS) !
              ! PREPARE VP TABLE ENTRIES FOR MM !
                        R2, #2 ! PRIDRITY !
0092 2102
             LD
0094 0002
0096 2105
             LD
                       R5. #OFF ! PREEMPT!
0000 8000
009A 2106
                       R6. #OFF ! KERNEL PROCESS!
             LD
009C 0000
              ! UPDATE VPT !
009E 5F00
             CALL
                      UPDATE_VP_TABLE ! (R1: DBR
00A0 01CA*
                                         R2: PRIORITY
                                         R5: PREEMPT FLAG
                                         R6: EXT_VP FLAG)
                                         RETURNS:
                                         R9: VP ID!
              ! INITIALIZE MM_CPU_TBL IN DISTRIBUTED MEMORY
               MANAGER WITH VP ID OF MM PROCESS !
              ! GET LOGICAL CPU # !
00A2 610A
             LD R10, PRDS.LOG CPU ID
00A4 0002*
```

```
LD MM_CPU_TBL (R10), R9
00A6 6FA9
*0000 8A00
             ! CREATE IDLE PROCESS !
00AA 2103
            LD
                     R3, #STACK BLOCK
00AC 0001
00AE 5F00
            CALL MM_ALLOCATE !R3: # OF BLOCKS
00B0 0000*
                                   RETURNS
                                   R2: START ADDR!
00B2 A121
            LD
                     R1, R2
00B4 2103
             LD
                     R3, #KERNEL FCW
00B6 5000
00B8 7604
            LDA
                     R4, IDLE_ENTRY
00BA 0000*
00BC 2105
            LD
                  R5. #%FFFF !NSP!
OOBE FFFF
00C0 7606
            LDA R6, PREEMPT_RET
00C2 0000*
00C4 93F1
            PUSH
                     @R15, R1 !SAVE STACK ADDR!
00C6 030F
            SUB
                     R15, #8
0008 0008
00CA 1CF9 LDM @R15, R3, #4
00CC 0303
            LD
                     RO, R15
OOCE A1FO
            ! INITIALIZE IDLE STACK VALUES !
00D0 5P00 CALL CREATE_STACK ! (RO: ARGUMENT PTR
00D2 0000*
                                     R1: TOP OF STACK
                                     R2-R14: INITIAL
                                     REG. STATES !
00D4 010F
          ADD R15, #8 !OVERLAY ARGUMENTS!
00D6 0008
            ! ALLOCATE MMU IMAGE FOR IDLE PROCESS !
00D8 5F00
            CALL ALLOCATE MMU ! RETURNS RO:DBR # !
00DA 0000*
            ! PREPARE IDLE PROCESS MMU ENTRIES !
            LD
                 R1, #STACK SEG ! SEG # !
00DC 2101
00DE 0001
           POP
                     R2, DR15 !GET STACK ADDR!
00E0 97F2
                     R3, #0
                                   ! WRITE ATTRIBUTE !
00E2 2103
           LD
00E4 0000
                     R4, #STACK_BLOCK-1 ! BLOCK LIMITS !
00E6 2104
           LD
00E8 0000
            ! SAVE DBR # !
00EA 93F0
                   aR 15, RO
            PUSH
            ! CREATE MMU IMAGE ENTRY !
           CALL UPDATE_MMU_IMAGE ! (R1: SEGMENT #
00EC 5F00
00EE 0000*
                                      R2: SEG ADDRESS
                                      R3: SEG ATTRIBUTES
```

```
R4: SEG LIMITS ) !
             ! RESTORE DBR # !
00F0 97F0
             POP
                      RO. DR15
             ! GET MMU ADDRESS !
                      GET_DBR_ADDR ! (RO: DBR #)
00F2 5F00
             CALL
00F4 0000*
                                        RETURNS
                                        (R1: DBR ADDRESS) !
             ! PREPARE VPT ENTRIES FOR IDLE PROCESS!
00F6 2102
             LD
                       R2, #0
                                       ! PRIORITY!
0000 8700
00FA 2105
                       R5, #OFF
                                      ! PREEMPT !
             LD
00FC 0000
00FE 2106
             LD
                      R6. #OFF
                                   ! KERNEL PROC!
0100 0000
             ! CREATE VPT ENTRIES !
                      UPDATE_VP_TABLE ! (R1: DBR
0102 5F00
             CALL
0104 01CA
                                         R2: PRIORITY
                                         R4: IDLE FLAG
                                         R5: PREEMPT
                                         R6: EXT_VP FLAG)
                                         RETURNS:
                                         R9: VP ID !
             ! ENTER VP ID OF IDLE PROCESS IN PRDS !
0106 6F09
             LD
                       PRDS.IDLE VP. R9
0108 0006*
             ! INITIALIZE IDLE VP'S !
010A 2102
             LD
                       R2, #1
                                      ! PRIORITY!
010C 0001
010E 2105
                       R5, #ON
             LD
                                      ! PREEMPT !
0110 FFFF
0112 2106
             LD
                      R6, #ON
                                !NON-KERNEL PROC!
0114 FFFF
0116 6100
             LD
                       RO, PRDS. VP_NR
0118 0004*
             ! INITIALIZE VP VALUES !
             DO
011A 5F00
             CALL
                  UPDATE_VP_TABLE ! (R1: DBR
011C 01CA
                                         R2: PRIORITY
                                         R4: IDLE FLAG
                                         R5: PREEMPT
                                         R6: EXT_VP FLAG)
                                         RETURNS:
                                        R9: VP ID !
011E AB00
             DEC
                       RO, #1
0120 OB00
             CP
                       RO, #G
0122 0000
0124 SEOE
            IF EQ !ALL VP'S INITIALIZED! THEN
0126 012C1
0128 5E08
                EXIT
```

```
012A 012E*
               FI
012C E8F6
              OD
              ! INITILIZE VPT HEADER !
              ! GET LOGICAL CPU NUMBER !
012E 6102
              LD
                          R2, PRDS.LOG_CPU_ID
0130 0002*
0132 4D05
              LD
                          VPT.LOCK, #OFF
0134 0000*
0136 0000
0138 4D25
              LD
                          VPT.RUNNING_LIST(R2), #NIL
013A 0002*
013C FFFF
013E 4D25
              LD
                          VPT.READY_LIST(R2), #NIL
0140 0006*
0142 FFFF
0144 4D08
              CLR
                          VPT.FREE_LIST !HEAD OF MSG LIST!
0146 000A*
           !THREAD VP'S BY PRIORITY AND SET STATES TO READY!
0148 8D28
                         R2 !START WITH VP #1!
              CLR
           THREAD:
               DO
014A 610D
                 LD
                          R13, PRDS.LOG_CPU_ID
014C 0002*
014E 76D3
                          R3, VPT. READY_LIST (R13)
                 LDA
0150 0006*
0152 7604
                 LDA
                          R4, VPT. VP.NEXT_READY_VP
0154 001C*
0156 7605
                          R5. VPT. VP.PRI
                 LDA
0158 0012*
015A 7606
                         R6, VPT. VP.STATE
                 LDA
015C 0014*
015E 2107
                         R7, #READY
                 LD
0160 0001
                 ! SAVE OBJ ID !
                          DR 15, R2
0162 93F2
                 PUSH
                 CALL
                          LIST_INSERT !R2: OBJ ID
0164 5F00
0166 0000*
                                         R3: LIST_HEAD_PTR ADDR
                                         R4: NEXT_OBJ PTR
                                         R5: PRIORITY PTR
                                         R6: STATE_PTR
                                         R7: STATE
                 ! RESTORE OBJ ID !
                          R2, @R15
0168 9772
                 POP
                          R2, #SIZEOF VP_TABLE
016A 0102
                 ADD
016C 0020
                          R2, # (NR_VP * (SIZEOF VP_TABLE))
016E 0B02
                 CP
0170 0100
                 IF EQ THEN EXIT FROM THREAD FI
0172 5E0E
0174 017A"
```

```
0176 5E08
0178 017C*
017A E8E7
             OD
             ! INITIALIZE VP MESSAGE LIST !
             ! NOTE: ONLY THE THREAD FOR THE MESSAGE
               LIST NEED BE CREATED AS ALL MESSAGES
               ARE INITIALLY AVAILABLE FOR USE. THE
               INITIAL MESSAGE VALUES WERE CREATED
               FOR CLARITY ONLY TO SHOW THAT THE
               MESSAGES HAVE NO USABLE INITIAL VALUE!
017C 8D18
             CLR
                         R 1
         MSG_LST_INIT:
             ! NOTE: R1 REPRESENTS CURRENT ENTRY IN
               MSG LIST, R2 REPRESENTS CURRENT POSITION
               IN MSG_LIST ENTRY, AND R3 REPRESENTS
               NEXT ENTRY IN MSG LIST. !
              DO
017E A112
               LD
                         R2, R1
0180 A123
               LD
                         R3, R2
                         R3, #SIZEOF MESSAGE
0182 0103
               ADD
0184 0010
              FILL_MSG:
                DO
0 186 4D25
                LD
                        VPT.MSG_Q.MSG(R2), #INVALID
0188 0110*
018A EEEE
018C A921
                INC
                         R2, #2
018E 8B32
                         R2, R3
                CP
0190 5E0E
                IF EQ THEN EXIT FROM FILL MSG FI
0192 01981
0194 5E08
0196 019A1
0198 E8F6
               OD
019A 4D15
               LD
                        VPT.MSG_Q.SENDER(R1), #NIL
019C 0120*
019E FFFF
01A0 A112
               LD
                         R2, R1
01A2 0101
               ADD
                        R1, #SIZEOF MSG_TABLE
01A4 0020
01A6 0B01
               CP
                         R1, #SIZEOF MSG_TABLE*NR_VP
01A8 0100
                IF EO
OIAA SEOE
                THEN
0 1AC 0 1BC'
01AE 4D25
                 LD
                         VPT.MSG_Q.NEXT_MSG(R2), #NIL
01B0 0122*
01B2 FFFF
0184 5E08
                EXIT FROM MSG_LST_INIT
01B6 01C2
01B8 5E08
               ELSE
0 1BA 01CO'
01BC 6F21
                 LD
                        VPT.MSG_Q.NEXT_MSG(R2), R1
```

01BE 0122* FI OD 01C0 E8DE ! GET LOGICAL CPU # FOR USE BY ITC GETWORK. ! R13, PRDS.LOG_CPU_ID 01C2 610D LD 01C4 0002* ! BOOTSTRAP COMPLETE ! ! START SYSTEM EXECUTION AT PREEMPT ENTRY ! ! POINT IN ITC GETWORK PROCEDURE ! 01C6 5E08 JP BOOTSTRAP_ENTRY

01C8 0000*
01CA END BOOTSTRAP

```
01CA
             UPDATE_VP_TABLE
                                   PROCEDURE
             * INITIALIZES VPT ENTRIES
             ************
             * REGISTER USE:
                                            *
                PARAMETERS:
                 R1: DBR ADDRESS
                                            *
                 R2: PRIORITY
                 R5: PREEMPT FLAG
                 R6: EXTERNAL VP FLAG
                RETURNS:
                 R9: ASSIGNED VP ID
               LOCAL VARIABLES:
                 R7: LOGICAL CPU #
                 R8: EXT VP LIST OFFSET
                 R9: VPT OFFSET
             ***************
             ENTRY
             ! GET OFFSET IN VPT FOR NEXT ENTRY !
                       R9. NEXT_AVAIL_VP
01CA 6109
             LD
01CC 0000'
01CE 6F91
             LD
                       VPT.VP.DBR(R9), R1
01D0 0010*
01D2 6F92
             LD
                       VPT.VP.PRI(R9), R2
01D4 0012*
01D6 6F96
                       VPT. VP. IDLE_FLAG(R9), R6
             LD
01D8 0016*
01DA 6F95
             LD
                       VPT.VP.PREEMPT(R9), R5
01DC 0018*
01DE 6107
             LD
                       R7, PRDS.LOG_CPU_ID
01E0 0002*
01E2 6F97
                       VPT. VP. PHYS PROCESSOR (R9) , R7
             LD
01E4 001A*
01E6 4D95
             LD
                       VPT.VP.NEXT_READY_VP(R9), #NIL
01E8 001C*
O1EA FFFF
01EC 4D95
             LD
                       VPT.VP.MSG_LIST(R9), #NIL
01EE 001E*
01FO FFFF
             ! CHECK EXTERNAL VP FLAG !
01F2 0B06
             CP
                       R6, #ON
01F4 FFFF
              IF EQ !EXTERNAL VP!
01F6 5E0E
              THEN ! VP IS TC VISIBLE !
01F8 0210 ·
01FA 6108
               LD
                       R8. NEXT EXT VP
01FC 0002'
               ! INSERT ENTRY IN EXTERNAL VP LIST !
01FE 6F89
               LD
                       EXT_VP_LIST (R8), R9
0200 0000*
0202 6F98
                       VPT.VP.EXT_ID(R9), R8
               LD
0204 0020*
```

```
INC R8, #2
0206 A981
0208 6F08
              LD
                       NEXT_EXT_VP, R8
020A 0002
020C 5E08
             ELSE ! VP BOUND TO KERNEL PROCESS!
020E 0216
0210 4D05
              LD
                       VPT.VP.EXT_ID, #NIL
0212 0020*
0214 FFFF
             FΙ
0216 A19A
             LD
                       R10, R9
0218 010A
             ADD
                       R10, #SIZEOF VP_TABLE
021A 0020
021C 6F0A
             LD
                       NEXT_AVAIL_VP, R10
021E 0000'
0220 9E08
              RET
            END UPDATE_VP_TABLE
0222
        END BOOTSTRAP_LOADER
```

Appendix F

LIBRARY FUNCTION LISTINGS

Z8000ASM 2.02 LOC OBJ CODE

STMT SOURCE STATEMENT

LIBRARY_FUNCTION MODULE

\$LISTON \$TTY

CONSTANT

KERNEL_FCW := %5000 STACK_SEG_SIZE := %100

STACK_BASE := STACK_SEG_SIZE-% 10 STATUS_REG_BLOCK:= STACK_SEG_SIZE-% 10

INTERRUPT_FRAME := STACK_BASE-4

INTERRUPT_REG := INTERRUPT_FRAME-34
N_S_P := INTERRUPT_REG-2

NIL := %FFFF

\$SECTION LIB_PROC GLOBAL

```
0000
            LIST_INSERT
                                   PROCEDURE
           * INSERTS OBJECTS INTO A LIST
            * BY ORDER OF PRIORITY AND SETS *
            * ITS STATE
            ************
            * REGISTER USE:
              PARAMETERS:
                R2: OBJECT ID
                R3: HEAD_OF_LIST_PTR ADDR
R4: NEXT_OBJ_PTR ADDR
                R5: PRIORITY PTR ADDR
                R6: STATE PTR ADDR
                R7: OBJECT STATE
            *
              LOCAL VARIABLES:
               R8: HEAD_OF_LIST_PTR
                R9: NEXT OBJ PTR
                R10: CURRENT_OBJ PRIORITY
                R11: NEXT OBJ PRIORITY
            **************
            ENTRY
            ! GET FIRST OBJECT IN LIST !
0000 2138
            LD
                           R8, DR3
0002 0B08
            CP
                           R8, #NIL
0004 FFFF
          IF EQ !LIST IS EMPTY! THEN
0006 5E0E
0008 0018
             ! PLACE OBJ AT HEAD OF LIST !
                           aR3, R2
000A 2F32
             LD
000C 7449
             LDA
                           R9, R4(R2)
000E 0200
                           aR9, #NIL
0010 0D95
            LD
0012 FFFF
            ELSE
0014 5E08
0016 005A'
             ! COMPARE OBJ PRI WITH LIST HEAD PRI !
                           R10. R5 (R2) !OBJ PRI!
0018 715A
             LD
001A 0200
                           R11, R5 (R8) ! HEAD PRI!
001C 715B
             LD
001E 0800
                           R10, R11
             CP
0020 8BBA
             IF GT !OBJ PRI>HEAD PRI! THEN
0022 5E02
0024 0030
                           DR3, R2 !PUT AT FRONT!
              LD
0026 2F32
                           R4(R2), R8
0028 7348
             LD
002A 0200
            ELSE ! INSERT IN BODY OF LIST !
002C 5E08
002E 005A
```

```
DO
                            R8, #NIL
                CP
0030 OB08
0032 FFFF
0034 SEOE
               IF EQ !END OF LIST! THEN
0036 003C'
0038 5E08
                EXIT FROM SEARCH_LIST
003A 0052*
                PΙ
003C 715B
                LD
                            R11, R5 (R8) !GET NEXT PRI!
003E 0800
0040 8BBA
                CP
                            R10, R11
0042 5E02
               IF GT !CUBBENT PRI>NEXT PRI! THEN
0044 004A*
0046 5E08
                EXIT FROM SEARCH_LIST
0048 00521
                FI
                ! GET NEXT OBJ !
004A A189
                LD
                            R9, R8
004C 7148
               LD
                            R8, R4(R9)
004E 0900
0050 E8EF
              OD ! END SEARCH_LIST !
               ! INSERT IN LIST !
0052 7348
              LD
                            R4(R2), R8
0054 0200
0056 7342
              LD
                            R4(R9), R2
0058 0900
             ΡI
             FI
             ! SET OBJECT'S STATE !
005A 7367
             LD
                            R6 (R2), R7
005C 0200
005E 9E08
            RET
0060
            END LIST INSERT
```

```
0060
              CREATE STACK
                                      PROCEDURE
              · ************************
              * INITIALIZES KERNEL STACK
              * SEGMENT FOR PROCESSES
              ***********
              * REGISTER USE:
                                              漱
                 PARAMETERS:
                                              *
                  RO: ARGUMENT POINTER
                   (INCLUDES: FCW, IC, NSP, AND
                                              *
                   RETURN POINT. SEE LOCAL
                   VARIABLES BELOW.)
                                              *
                  R1: TOP OF STACK
                  R2-R14: INITIAL REGISTER
                   STATES. (NOTE: IN DEMO, NO*
                   SPECIFIC INITIAL REGISTER *
                   VALUES ARE SET, EXCEPT R13*
                   (USER ID) FOR USER PRO-
                   CESSES.)
              ******************
                 LOCAL VARIABLES
                 (FROM ARGUMENTS STORED ON
                  STACK.)
                  R3: FCW
                  R4: PROCESS ENTRY POINT (IC) *
                  R5: NSP
                  R6: PREEMPT RETURN POINT
              **************
              ENTRY
0060 93F0
                        DR 15, RO !SAVE ARGUMENT PTR!
              PUSH
0062 ADF0
                        RO, R15
              EΧ
                                !SAVE SP!
0064 341F
              LDA
                        R15, R1 (#INTERRUPT REG)
0066 00CA
0068 1CF9
              LDM
                        @R 15, R1, #16
                                       !INITIAL REG. VALUES!
006A 010F
              ! NOTE: ONLY REGISTERS R2-R14 MAY CONTAIN
                INITIALIZATION VALUES !
006C A10F
                        R15, RO !RESTORE SP!
              LD
                        RO, DR15 !RESTORE ARGUMENT PTR!
006E 97F0
              POP
0070 A1FE
              LD
                        R14, R15 !SAVE CALLER RETURN POINT!
0072 A10F
                        R15, RO !GET ARGUMENT PTR!
              LD
0074 1CF1
                        R3, DR15, #4 !LOAD ARGUMENTS!
              LDM
0076 0303
0078 341F
              LDA
                        R15, R1(#INTERRUPT_FRAME)
007A 00EC
007C 1CF9
                        DR 15, R3, #2 !INIT IRET FRAME!
              LDM
007E 0301
0080 341F
                        R15, R1 (#N_S_P)
              LDA
0082 00C8
0084 2FF5
                        aR15, R5
                                  !SET NSP!
              LD
                        R15, #2
              SUB
0086 030F
0088 0002
008A 2FF6
                        DR15, R6 !PREEMPT RET POINT!
              LD
008C 3418
              LDA
                        R8, R1(#STACK_BASE)
```

```
008E 00F0
              ! INITIALIZE STATUS REGISTER BLOCK !
0090 2100
                        RO, #KERNEL_FCW
              LD
0092 5000
0094 1C89
                        ars, R15, #2 !SAVE SP & FCW!
             LDM
0096 0F01
                        R15, R14 ! RESTORE RETURN POINT!
0098 A1EF
             LD
009A 9E08
              RET
009C
            END CREATE_STACK
        END LIBRARY_FUNCTION
```

Appendix G

INNER TRAFFIC CONTROLLER LISTINGS

Z8000ASM 2.02

LOC OBJ CODE STMT SOURCE STATEMENT

INNER_TRAFFIC_CONTROL MODULE

\$LISTON \$TTY

!**1. GETWORK:

- A. NORMAL ENTRY DOES NOT SAVE REGISTERS.
 (THIS IS A FUNCTION OF THE GATEKEEPER).
- B. R14 IS AN INPUT PARAMETER TO GETWORK THAT SIMULATES INFO THAT WILL EVENTUALLY BE ON THE MMU HARDWARE. THIS REGISTER MUST BE ESTABLISHED AS A DBR BY ANY PROCEDURE INVOKING GETWORK.
- C. THE PREEMPT INTERRUPT ENTRY HANDLER DOES NOT USE THE GATEKEEPER AND MUST PERFORM FUNCTIONS NORMALLY ACCOMPLISHED BY IT PRIOR TO NORMAL ENTRY AND EXIT.

 (SAVE/RESTORE: REGS, NSP: UNLOCK VPT, TEST INT)

2. GENERAL:

- A. ALL VIOLATIONS OF VIRTUAL MACHINE INSTRUCTIONS ARE CONSIDERED ERROR CONDITIONS AND WILL RETURN SYSTEM TO THE MONITOR WITH AN ERROR CODE IN ROAND THE PC VALUE IN R1.
- B. ITC PROCEDURES CALLING GETWORK PASS DBR (REGISTER R14) AND LOGICAL CPU NUMBER (REGISTER R13) AS INPUT PARAMETERS. (INCLUDES: SIGNAL, WAIT, SWAP_VDBR, PHYS_PREEMPT_HANDLER, AND IDLE). !

CONSTANT

! ****** ERROR CODES ******* ! := 0 ! UNAUTHORIZED LOCK! UL := 1 ! MESSAGE LIST EMPTY ! M_L_EM ! MESSAGE LIST ERROR ! :≃ 2 M_L_ER := 3 ! READY LIST EMPTY ! R_L_E := 4 ! MESSAGE LIST OVERFLOW ! MLO ! SWAP NOT ALLOWED ! := 5 SNA := 6 ! VP INDEX ERROR ! V_I_E := 7 ! MMU UNAVAILABLE ! M_U

```
! ****** SYSTEM PARAMETERS *******!
   NR SDR
                   := 64
                          !LONG WORDS!
                   := 2
   NR_CPU
   NR VP
                   := NR CPU*4
   NR_AVAIL_VP
                  := NR_CPU*2
   MAX_DBR_NR
                   := 10 !PER CPU!
   STACK SEG
                  := 1
   PRDS SEG
                  := 0
   STACK_SEG_SIZE := %100
   ! **** OFFSETS IN STACK SEG *****!
   STACK_BASE
                  := STACK_SEG_SIZE-%10
   STATUS_REG_BLOCK: = STACK_SEG_SIZE-%10
   INTERRUPT_FRAME := STACK_BASE-4
                 := INTERRUPT_FRAME-34
   INTERRUPT_REG
                   := INTERRUPT_REG-2
   N_S_P
   F_C_W
                   := STACK_SEG_SIZE-%E
   ON
          := %FFFF
   OFF
          := 0
   RUNNING := 0
          := 1
   READY
   WAITING := 2
          := %FFFF
   INVALID := %EEEE
   MONITOR := %A900
                         ! HBUG ENTRY!
   KERNEL FCW := %5000
   AVAILABLE := 0
   ALLOCATED := %FF
TYPE
   MESSAGE ARRAY [ 16 BYTE ]
   ADDRESS WORD
   VP_INDEX
                   INTEGER
   MSG INDEX
                  INTEGER
   SEG_DESC_REG
                RECORD
               [
                 BASE ADDRESS
                 ATTRIBUTES
                                  BYTE
                 LIMITS
                                   BYTE
                1
   UMM
                   ARRAY[NR_SDR SEG_DESC_REG]
   MSG TABLE RECORD
   [ MSG
                   MESSAGE
     SENDER
                   VP_INDEX
     NEXT_MSG
                   MSG INDEX
     FILLER
                  ARRAY [6, WORD]
    ]
```

```
VP_TABLE RECORD
                 DBR ADDRESS
                  PRI
                                WORD
                  STATE
                                WORD
                  IDLE_FLAG
                                WORD
                  PREEMPT
                                WORD
                  PHYS_PROCESSOR WORD
                  NEXT_READY_VP VP_INDEX
                  MSG_LIST
                               MSG_INDEX
                  EXT ID
                                WORD
                  FILLER 1
                                ARRAY[ 7, WORD ]
           EXTERNAL
             LIST_INSERT
                              PROCEDURE
           GLOBAL
             BOOTSTRAP_ENTRY LABEL
           $SECTION ITC_DATA
0000
             VPT
                     RECORD
               [ LOCK
                              WORD
                 RUNNING_LIST ARRAY[NR_CPU WORD]
                 READY_LIST ARRAY[NR_CPU WORD]
FREE_LIST MSG_INDEX
                 VIRT_INT_VEC ARRAY[1, ADDRESS]
                 FILLER_2
                              WORD
                 VP.
                           ARRAY [NR_VP, VP_TABLE]
                 MSG Q
                              ARRAY [ NR_VP, MSG_TABLE ]
0210
                         ARRAY[NR_AVAIL_VP WORD]
             EXT_VP_LIST
          $SECTION MMU_DATA
0000
             MMU_IMAGE
                            RECORD
                 HMU STRUCTURE ARRAY[MAX_DBR_NR MMU]
             NEXT_AVAIL_MMU ARRAY[MAX_DBR_NR BYTE]
0 A 0 0
OAOA
             PRDS
                    RECORD
                  [PHYS_CPU_ID WORD
                   LOG_CPU_ID INTEGER
                   VP NR
                               WORD
                           VP_INDEX]
                   IDLE VP
```

```
$SECTION ITC_INT_PROC
            INTERNAL
                                  PROCEDURE
0000
            GETWORK
            ! **********************
            * SWAPS VIRTUAL PROCESSORS
            * ON PHYSICAL PROCESSOR.
            ***************
            * PARAMETERS:
            * R13: LOGICAL CPU #
            * REGISTER USE:
              STATUS REGISTERS
                R14: DBR (SIMULATION)
                R15: STACK_POINTER
            *
              LOCAL VARIABLES:
                R1: READY VP (NEW)
                R2: CURRENT_VP (OLD)
            *
                R3: FLAG CONTROL WORD
                R4: STACK_SEG BASE ADDR
                R5: STATUS_REG_BLOCK ADDR *
                R6: NORMAL STACK POINTER
            ************
            ENTRY
            ! GET STACK BASE !
0000 31E4
            LD
                      R4, R14 (#STACK_SEG*4)
0002 0004
0004 3445
            LDA
                      R5, R4 (#STATUS REG BLOCK)
0006 00F0
            ! * * SAVE SP * * !
0008 2F5F
            LD
                      DR5, R15
            ! * * SAVE PCW * * !
000A 7D32
                      R3, FCW
            LDCTL
000C 3343
            LD
                      R4 (#F_C W) , R3
000E 00F2
        BOOTSTRAP_ENTRY:
                                ! GLOBAL LABEL !
            ! GET READY_VP LIST !
0010 61D1
                    R1, VPT. READY LIST (R13)
            LD
0012 0006
           SELECT_VP:
            DO ! UNTIL ELGIBLE READY VP FOUND !
0014 4D11
             CP VPT. VP. IDLE FLAG (R1) . #ON
0016 0016
0018 FFFF
001A 5E0E
            IF EO ! VP IS IDLE ! THEN
001C 0030*
001E 4D11
             CP VPT.VP.PREEMPT(R1), #ON
0020 0018
0022 FFFF
0024 5E0E
             IF EQ ! PREEMPT INTERRUPT IS ON ! THEN
0026 002C*
0028 5E08
               EXIT FROM SELECT VP
002A 003C*
```

```
FI
002C 5E08
              ELSE ! VP NOT IDLE !
002E 0034
0030 5E08
                EXIT FROM SELECT VP
0032 003C
              ΡI
              ! GET NEXT READY_VP !
0034 6113
              LD R3, VPT. VP. NEXT READY VP(R1)
0036 001C
0038 A131
             LD
                  R1, R3
003A E8EC
             OD
            ! NOTE: THE READY_LIST WILL NEVER BE EMPTY SINCE
                THE IDLE VP, WHICH IS THE LOWEST PRI VP.
                WILL NEVER BE REMOVED FROM THE LIST.
                IT WILL RUN ONLY IF ALL OTHER READY VP'S ARE
                IDLING OR IF THERE ARE NO OTHER VP'S ON
                THE READY_LIST. ONCE SCHEDULED, IT
                WILL RUN UNTIL RECEIVING A HOWE INTERRUPT. !
            ! NOTE: R14 IS USED AS DBR HERE. WHEN MMU
                IS AVAILABLE THIS SERIES OF SAVE AND LOAD
                INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O
                INSTRUCTIONS TO THE MMU. !
            ! PLACE NEW_VP IN RUNNING STATE !
            LD VPT. VP. STATE (R1), #RUNNING
003C 4D15
003E 0014
0040 0000
0042 6FD1
             LD
                 VPT.RUNNING_LIST(R13), R1
0044 0002
             ! * * SWAP DBR * * !
0046 611E
             LD R14, VPT.VP.DBR (R1)
0048 0010
             ! LOAD NEW_VP SP !
004A 31E4
                 R4, R14 (#STACK_SEG*4)
             LD
004C 0004
004E 3445
             LDA R5, R4 (#STATUS_REG_BLOCK)
0050 00F0
             LD R15. @R5
0052 215F
             ! * * LOAD NEW FCW * * !
0054 3143
             LD R3, R4 (#F_C_W)
0056 00F2
             LDCTL FCW, R3
0058 7D3A
005A 9E08
            RET
```

END GETWORK

005C

```
005C
                                  PROCEDURE
           ENTER_MSG_LIST
           * INSERTS POINTER TO MESSAGE
           * FROM CURRENT_VP TO SIGNALED_VP*
           * IN FIFO MSG_LIST
           ************
           * REGISTER USE:
              PARAMETERS:
               R8 (R9): MSG (INPUT)
           *
               R1: SIGNALED_VP (INPUT)
               R13: LOGICAL CPU NUMBER
           *
             LOCAL VARIABLES:
           *
               R2: CURRENT VP
           *
               R3: FIRST_FREE_MSG
               R4: NEXT_FREE_MSG
               R5: NEXT_Q_MSG
           *
               R6: PRESENT_Q_MSG
           ***********
           ENTRY
005C 61D2
            LD R2, VPT.RUNNING_LIST(R13)
005E 0002*
            ! GET FIRST MSG PROM FREE LIST !
0060 6103
            LD R3, VPT.FREE_LIST
0062 000A'
                ! * * * * DEBUG * * * * !
0064 0303
                CP R3, #NIL
0066 FFFF
0068 5E0E
               IF EQ THEN
006A 0078
006C 7601
                LDA R1, $
006E 006C
0070 2100
                LD RO, #M L_O! MESSAGE LIST OVERFLOW!
0072 0004
0074 5F00
                 CALL MONITOR
0076 A900
                PI
                ! * * * END DEBUG * * * !
0078 6134
            LD R4, VPT.MSG_Q.NEXT_MSG(R3)
007A 0122'
007C 6F04
            LD VPT.FREE_LIST, R4
007E 000A
            ! INSERT MESSAGE LIST INFORMATION !
0080 763A
            LDA
                      R10, VPT. MSG_Q. MSG(R3)
0082 0110
0084 2107
            LD
                      R7, #SIZEOF MESSAGE
0086 0010
0088 BA81
            LDIRB
                     @R10,@R8,R7
008A 07A0
008C 6F32
            LD VPT.MSG_Q.SENDER(R3), R2
008E 0120
```

```
! INSERT MSG IN MSG_LIST !
0090 6115
             LD R5, VPT. VP. MSG_LIST(R1)
0092 001E
0094 0B05
           CP R5, #NIL
0096 FFFF
             IF EQ ! MSG LIST IS EMPTY! THEN
0098 5E0E
009A 00A4
             ! INSERT MSG AT TOP OF LIST !
009C 6F13
             LD VPT. VP. MSG LIST(R1), R3
009E 001E
00A0 5E08
             ELSE ! INSERT MSG IN LIST !
00A2 00BC
             MSG_Q_SEARCH:
             DO ! WHILE NOT END OF LIST !
00A4 0B05
             CP
                        R5, #NIL
00A6 FFFF
00A8 5E0E
             IF EQ ! END OF LIST! THEN
00AA 00B0
00AC 5E08
              EXIT FROM MSG_Q_SEARCH
00AE 00B8
              FI
              ! GET NEXT LINK !
00B0 A156
                     R6, R5
             LD
                     R5. VPT.MSG_Q.NEXT_MSG(R6)
00B2 6165
             LD
00B4 0122*
00B6 E8F6
             OD
             ! INSERT MSG IN LIST !
                     VPT.MSG_Q.NEXT_MSG(R6), R3
00B8 6F63
            LD
00BA 0122
             FI
                      VPT.MSG_Q.NEXT_MSG(R3), R5
00BC 6F35
             LD
00BE 0122
00C0 9E08
            RET
         END ENTER_MSG_LIST
00C2
```

```
00C2
            GET FIRST MSG
                                    PROCEDURE
           * REMOVES MSG FROM MSG_LIST
            * AND PLACES ON FREE LIST.
                                            *
            * RETURNS SENDER'S MSG AND
            * VP ID
            ************
            *REGISTER USE:
            * PARAMETERS:
              R8 (R9): MSG POINTER (INPUT)
              R13: LOGICAL CPU NUMBER (INPUT) *
            * R1: SENDER VP (RETURNED)
            * LOCAL VARIABLES
              R2: CURRENT VP
              R3: FIRST_MSG
              R4: NEXT_MSG
               R5: NEXT FREE MSG
            * R6: PRESENT FREE MSG
            *************
            ENTRY
00C2 61D2
            LD
                     R2. VPT.RUNNING_LIST (R13)
00C4 0002*
            ! REMOVE FIRST MSG FROM MSG LIST !
00C6 6123
           LD
                     R3, VPT. VP.MSG_LIST(R2)
00C8 001E'
                     ! * * * * DEBUG * * * * !
00CA 0B03
                     CP R3, #NIL
OOCC FFFF
OOCE SECE
                    IF EQ THEN
OODO OODE
00D2 2100
                     LD RO, #M_L_EM ! MSG LIST EMPTY !
00D4 0001
00D6 7601
                     LDA R1, $
00D8 00D6 ·
00DA 5F00
                     CALL MONITOR
00DC A900
                     FI
                     ! * * * END DEBUG * * * !
00DE 6134
            LD
                     R4. VPT. MSG_Q.NEXT MSG (R3)
00E0 0122 º
00E2 6F24
            LD
                     VPT. VP.MSG_LIST(R2), R4
00E4 001E*
                    ! INSERT MESSAGE IN FREE_LIST !
00E6 6105
            LD
                     R5, VPT.FREE_LIST
00E8 000A .
00EA 0B05
                     R5, #NIL
            CP
OOEC FFFF
OOEE SEOE
            IF EQ
                   ! FREE_LIST IS EMPTY !
                                          THEN
00F0 0100'
            ! INSERT AT TOP OF LIST!
00F2 6F03
                     VPT.FREE_LIST, R3
            LD
00F4 000A*
```

```
00F6 4D35
           LD
                      VPT.MSG_Q.NEXT_MSG(R3), #NIL
00F8 01221
OOFA FFFF
00FC 5E08
            ELSE ! INSERT IN LIST !
OOFE 011C'
           FREE_Q_SEARCH:
              DO
                      R5, #NIL
0100 OB05
              CP
0102 FFFF
0104 5E0E
             IF EQ ! END OF LIST ! THEN
0106 010C'
0108 5E08
                EXIT FROM FREE Q SEARCH
010A 01141
               FI
               ! GET NEXT MSG !
010C A156
               LD
                       R6, R5
010E 6165
              LD
                      R5, VPT.MSG_Q.NEXT_MSG (R6)
0110 0122
0112 E8F6
             OD
             ! INSERT IN LIST !
0114 6F63
             LD
                      VPT.MSG_Q.NEXT_MSG(R6), R3
0116 0122
0118 6F35
             LD
                      VPT.MSG Q.NEXT MSG(R3), R5
011A 0122'
             FI
             ! GET MESSAGE INFORMATION:
               (RETURNS R1: SENDING VP)
011C 6131
                      R1, VPT.MSG Q.SENDER (R3)
011E 0120'
0120 763A
             LDA
                      R10, VPT. MSG_Q. MSG(R3)
0122 0110
0124 2107
            LD
                   R7, #SIZEOF MESSAGE
0 1 2 6 0 0 1 0
0128 BAA1
                      aR8, aR10, R7
            LDIRB
012A 0780
012C 9E08
            RET
           END GET_FIRST_MSG
012E
```

```
* * INNER TRAFFIC CONTROL ENTRY POINTS * * !
           ! NOTE: ALL INTERRUPTS MUST BE MASKED WHENEVER
             THE VPT IS LOCKED. THIS IS TO PREVENT AN
             EMBRACE FROM OCCURRING SHOULD AN INTERRUPT
             OCCUR WHILE THE VPT IS LOCKED. !
           GLOBAL
           $SECTION ITC_GLB_PROC
           PREEMPT_RET LABEL
           KERNEL EXIT LABEL
           CREATE INT VEC
0000
                                 PROCEDURE
          · **********************
           * CREATES ENTRY IN VIRTUAL INT-*
           * ERRUPT VECTOR WITH ADDRESS
           * OF THE VIRTUAL INTERRUPT HAN-*
           * DLER.
           *********
           * PARAMETERS:
              R1: VIRTUAL INTERRUPT #
              R2: INTERRUPT HANDLER ADDR
                                         *
           ************
           ENTRY
            ! COMPUTE OFFSET IN VIRTUAL
              INTERRUPT VECTOR !
0000 1900
            MULT
                      RRO, #SIZEOF ADDRESS
0002 0002
            ! SAVE ADDRESS OF VIRTUAL INTERRUPT
```

END CREATE INT VEC

LD

RET

0004 6F12

000A

0006 000C* 0008 9E08 HANDLER IN INTERRUPT VECTOR !

VPT.VIRT_INT_VEC(R1), R2

```
GET_DBR_ADDR PROCEDURE
000A
         * CALCULATES DBR ADDRESS FROM *
         * DBR NUMBER
         **********
         * REGISTER USE:
           PARAMETERS:
            RO: DBR #
                                   *
           RETURNS:
                                   *
            R1: DBR ADDRESS
         ***********
         ENTRY
          ! GET BASE ADDRESS OF MMU IMAGE !
000A 7601
          LDA
                   R1, MMU_IMAGE
000C 0000°
          ! ADD DBR HANDLE (OFFSET) TO MMU BASE
            ADDRESS TO OBTAIN DBR ADDRESS !
                  R1, R0
000E 8101
0010 9E08
          RET
0012
         END GET_DBR_ADDR
```

```
0012
            ALLOCATE MMU
                                   PROCEDURE
           * ALLOCATES NEXT AVAILABLE MMU *
            * IMAGE AND CREATES PRDS ENTRY *
            *********
            * REGISTER USE:
               RETURNS:
               RO: DBR #
            *
               LOCAL VARIABLES:
            *
               R1: SEGMENT #
                R2: PRDS ADDRESS
                R3: PRDS ATTRIBUTES
               R4: PRDS LIMITS
            ************
            ENTRY
             ! GET NEXT AVAILABLE DBR # !
0012 8D08
             CLR
                       R<sub>0</sub>
0014 8D18
             CLR
                       R 1
             ! NOTE: THE POLLOWING IS A SAFE SEQUENCE
               AS NEXT_AVAIL_MMU AND MMU ARE CPU LOCAL!
         GET DBR:
             DO
0016 4C11
               CPB
                       NEXT_AVAIL_MMU (R1) , #AVAILABLE
0018 0A00 1
001A 0000
               IF EO
                      !MMU ENTRY IS AVAILABLE!
001C 5E0E
                 THEN
001E 002E
0020 4C15
                 LDB
                       NEXT_AVAIL_MMU (R1) , #ALLOCATED
0022 0A00 ·
0024 PEFF
0026 5E08
                 EXIT FROM GET DBR
0028 004A*
002A 5E08
                ELSE
                       !CURRENT ENTRY IS ALLOCATED!
002C 0048
002E A910
                 INC
                       R1, #1
0030 0100
                  ADD
                       RO. #SIZEOF MMU
0032 0100
                   ! * * * * DEBUG * * * * !
0034 0B01
                  CP R1, #MAX_DBR_NR
0036 000A
0038 5E0E
                  IF EQ THEN
003A 0048*
003C 2100
                     LD
                               RO, #M_U !MMU UNAVAILABLE!
003E 0007
0040 7601
                    LDA
                              R1, 3
0042 0040*
0044 5F00
                    CALL
                               MONITOR
0046 A900
                  FI
                   ! * * * END DEBUG * * * !
               FI
0048 E8E6
            OD
```

```
004A 2101 LD R1, #PRDS_SEG ! SEGMENT NO. !
004C 0000
                    R2, PRDS ! PRDS ADDR !
004E 7602
          LDA
0050 OAOA'
0052 2103
                    R3, #1 ! READ ATTR !
           LD
0054 0001
0056 2104
        LD
                    R4, #((SIZEOF PRDS) - 1) /256
0058 0000
           ! PRDS LIMITS !
           ! CREATE PRDS ENTRY IN MMU IMAGE !
005A 5F00
          CALL UPDATE MMU IMAGE ! (R1: SEGMENT #
005C 00601
                                       R2: SEG ADDRESS
                                       R3: ATTRIBUTES
                                       R4: SEG LIMITS)!
005E 9E08 RET
     END ALLOCATE_MMU
0060
```

```
UPDATE_MMU_IMAGE
                               PROCEDURE
0060
          * CREATES SEGMENT DESCRIPTOR
           * ENTRY IN MMU IMAGE
           **********
           * REGISTER USE:
                                        *
             PARAMETERS:
              RO: DBR #
                                        *
              R1: SEGMENT #
              R2: SEGMENT ADDRESS
              R3: SEGMENT ATTRIBUTES
              R4: SEGMENT LIMITS
             LOCAL VARIABLES:
              R10: MMU BASE ADDRESS
                                        *
              R13: OFFSET VARIABLE
           ***********
           ENTRY
0060 210A
           LD R10, #MMU_IMAGE ! MMU BASE ADDRESS !
0062 0000
0064 810A
            ADD R10, RO
0066 210D
            LD R13, #SIZEOF SEG DESC REG
0068 0004
006A 991C
            MULT RR12, R1 ! COMPUTE SEG_DESC OFFSET !
            ADD R10, R13 !ADD OFFSET TO BASE ADDRESS!
006C 81DA
            ! INSERT DESCRIPTOR DATA!
006E 2FA2
            LD 0R10, R2
0070 A9A1
            INC R10, #2
0072 ODA8
           CLR DR10
0074 2EAC
               DR 10, RL4
           LDB
0076 A9A0
           INC
               R10, #1
0078 20AC
           LDB RL4, DR10
007A 0A0B
           CPB RL3, #%(2)00001000 ! EXECUTE !
007C 0808
007E 5E0E
          IF EQ THEN
0080 008A ·
0082 060C
               ANDB RL4, #% (2) 11110111 ! EXECUTE MASK!
0084 F7F7
0086 5E08
           ELSE
0088 008E
008A 060C
               ANDB RL4, #%(2) 11111110 ! READ MASK!
008C FEFE
            FI
008E 84BC
           ORB RL4, RL3
0090 2EAC
           LDB @R10, RL4
0092 9E08
           RET
0094
        END UPDATE_MMU_IMAGE
```

```
0094
            WAIT
                                     PROCEDURE
           * INTRA_KERNEL SYNC/COM PRIMATIVE *
            * INVOKED BY KERNEL PROCESSES
            ************
            * PARAMETERS
              R8 (R9): MSG POINTER (INPUT)
              R1: SENDING_VP (RETURN)
            * GLOBAL VARIABLES
             R14: DBR (PARAM TO GETWORK)
           * LOCAL VARIABLES
              R2: CURRENT_VP (RUNNING)
              R3: NEXT_READY_VP
R4: LOCK_ADDRESS
              R13: LOGICAL CPU NUMBER
           *************
           ENTRY
            ! MASK INTERRUPTS !
0094 7C01
            DI
                  VI
            ! LOCK VPT !
0096 7604
                      R4. VPT. LOCK
            LDA
0098 00001
009A 5F00
            CALL
                      SPIN_LOCK ! (R4:¬VPT.LOCK) !
009C 0282
            ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
            ! GET CPU NUMBER !
009E 5F00
            CALL
                      GET_CPU_NO !RETURNS:
00A0 02C8
                                   R1:CPU #
                                   R2: # VP S!
00A2 A11D
            LD
                      R13, R1
                      R2, VPT.RUNNING_LIST (R 13)
00A4 61D2
            LD
00A6 0002 ·
00A8 6123
                      R3, VPT. VP.NEXT_READY_VP(R2)
            LD
00AA 001C
                      VPT. VP. MSG LIST(R2), #NIL
00AC 4D21
            CP
00AE 001E
OOBC FFFF
             IF EQ ! CURRENT VP'S MSG LIST IS EMPTY ! THEN
00B2 5E0E
00B4 00EA .
             ! REMOVE CURRENT_VP FROM READY_LIST !
                       ! * * * * DEBUG * * * * !
                              R3, #NIL
00B6 0B03
00B8 FFFF
OOBA SEOE
                      IF EQ THEN
OOBC OOCA'
                       LD RO, #R_L_E ! READY LIST EMPTY !
00BE 2100
00C0 0003
                       LDA R1. $
00C2 7601
00C4 00C2
                       CALL MONITOR
00C6 5F00
00C8 A900
```

```
FI
                       ! * * * END DEBUG * * * !
OOCA 6FD3
              LD
                      VPT.READY_LIST(R13), R3
00CC 0006 1
00CE 4D25
                      VPT.VP.NEXT_READY_VP(R2), #NIL
               LD
00D0 001C1
00D2 FFFF
               ! PUT IT IN WAITING STATE !
00D4 4D25
               LD VPT. VP. STATE (R2), #WAITING
00D6 0014*
00D8 0002
               ! SET DBR !
00DA 612E
              LD
                      R14, VPT.VP.DBR(R2)
00DC 00101
               ! SCHEDULE FIRST ELGIBLE READY VP !
00DE 93F8
               PUSH
                         @R15,R8
               ! SAVE LOGICAL CPU # !
                       @R15, R13
00E0 93FD
               PUSH
00E2 5F00
                       CALL GETWORK !R13:CPU #
00E4 0000 *
                                         R14:DBR!
              ! RESTORE CPU # !
00E6 97FD
              POP R13, @R15
00E8 97F8
              POP
                        R8.0R15
             FI
             ! GET FIRST MSG ON CURRENT VP'S MSG LIST !
00EA 5F00
             CALL GET_FIRST_MSG ! COPIES MSG IN MSG ARRAY!
00EC 00C2*
                                 ! R13: LOGICAL CPU # !
                                 !RETURNS R1: SENDER VP !
             ! UNLOCK VPT !
00EE 4D08
            CLR VPT.LOCK
00F0 0000°
            ! UNMASK VECTORED INTERRUPTS !
00F2 7C05
            EI VI
             ! RETURN: R1:SENDER_VP !
00F4 9E08
            RET
00F6 END WAIT
```

```
00F6
                                      PROCEDURE
           * INTRA_KERNEL SYNC /COM PRIMATIVE *
            * INVOKED BY KERNEL PROCESSES
            ************
            * REGISTER USE:
               PARAMETERS:
                R8 (R9): MSG POINTER (INPUT)
                R1: SIGNALED VP_ID (INPUT)
            * GLOBAL VARIABLES
               R13: CPU # (PARAM TO GETWORK)
               R14: DBR (PARAM TO GETWORK)
               LOCAL VARIABLES:
               R1: SIGNALED VP
                R2: CURRENT_VP
                R4: VPT.LOCK ADDRESS
            **************
            ENTRY
             ! SAVE VP ID !
00F6 93F1
                    @R15, R1
             PUSH
             ! MASK INTERRUPTS !
00F8 7C01
             DI
                  VI
             ! LOCK VPT !
00FA 7604
             LDA
                      R4, VPT.LOCK
00FC 0000 *
00FE 5F00
                      SPIN_LOCK ! (R4:-VPT.LOCK) !
            CALL
0100 0282
            ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
             ! GET LOGICAL CPU # !
0102 5F00
                      GET_CPU_NO !RETURNS:
             CALL
0104 02C8 ·
                                  R1:CPU #
                                  R2:# VP*S!
0106 A11D
                      R13, R1
             LD
             ! RESTORE VP ID !
0108 97F1
                      R1, @R15
            POP
             ! PLACE MSG IN SIGNALED_VP'S MSG_LIST !
            CALL ENTER_MSG_LIST ! (R8:MSG POINTER
010A 5F00
010C 005C*
                                  R1:SIGNALED_VP
                                  R13:LOGICAL CPU #)!
                      VPT. VP. STATE (R1) . #WAITING
010E 4D11
            CP
0110 00141
0112 0002
0114 SEOE
            IF EQ ! SIGNALED_VP IS WAITING !
                                               THEN
0116 01481
               ! WAKE IT UP AND MAKE IT READY !
0118 A112
                      R2, R1
              LD
                      R3, VPT.READY_LIST(R13)
011A 76D3
              LDA
011C 0006'
                      R4. VPT. VP.NEXT_READY_VP
011E 7604
              LDA
0120 001C*
```

```
0122 7605
              LDA R5, VPT. VP.PRI
0124 00121
0126 7606
              LDA
                      R6. VPT. VP.STATE
0128 0014
                      R7, #READY
012A 2107
              LD
012C 0001
              ! SAVE LOGICAL CPU # !
                     @R15, R13
012E 93FD
              PUSH
0130 5F00
              CALL
                     LIST INSERT !R2: OBJ ID
0132 0000*
                                    R3: LIST_PTR ADDR
                                    R4: NEXT_OBJ_PTR
                                    R5: PRIORITY_PTR
                                    R6: STATE_PTR
                                    R7: STATE !
              ! RESTORE LOGICAL CPU # !
0134 97FD
              POP
                     R13, DR15
             ! PUT CURRENT_VP IN READY_STATE !
0136 61D2
             LD
                      R2, VPT.RUNNING_LIST (R 13)
0138 0002
013A 4D25
            LD
                      VPT. VP. STATE (R2) . #READY
013C 00141
013E 0001
             ! SET DBR !
0140 612E
            LD
                      R14, VPT.VP.DBR(R2)
0142 0010
             ! SCHEDULE FIRST ELGIBLE READY VP !
            CALL GETWORK !R13:LOGICAL CPU #
0144 5F00
0146 0000 *
                               R14: DBR !
            FI
            ! UNLOCK VPT !
0148 4D08
            CLR VPT.LOCK
014A 0000°
            ! UNMASK VECTORED INTERRUPTS !
014C 7C05
            EI VI
014E 9E08
           RET
0150 END SIGNAL
```

```
0150
           SET_PREEMPT
                              PROCEDURE
           ! **********************
           * SETS PREEMPT INTERRUPT ON*
           * TARGET_VP. CALLED BY TC_ *
           * ADVANCE.
           ************
           * REGISTER USE:
           * PARAMETERS:
             R1:TARGET_VP_ID (INPUT)
           * LOCAL VARIABLES
              R1: VP_INDEX
           ***********
           ENTRY
             ! NOTE: DESIGNED AS SAFE SEQUENCE SO VPT NEED
              NOT BE LOCKED. !
             ! CONVERT VP_ID TO VP INDEX !
0150 6112
            LD
                      R2, EXT_VP_LIST(R1)
0152 02101
             ! TURN ON TGT_VP PREEMPT FLAG !
0154 4D25
                      VPT. VP. PREEMPT (R2) , #ON
            LD
0156 0018
0158 FFFF
             ! ** IF TARGET VP NOT LOCAL
                   ( NOT BOUND TO THIS CPU )
            [IE, IF <<CPU_SEG>>CPU_ID<>VPT.VP.PHYS_CPU(R1)]
            THEN SEND HARDWARE PREEMPT INTERRUPT TO
              VPT.VP.CPU(R1). **!
015A 9E08
            RET
015C
         END SET_PREEMPT
```

```
0.15C
           IDLE
                          PROCEDURE
          ****************
           * LOADS IDLE DBR ON
           * CURRENT VP. CALLED BY *
           * TC GETWORK.
           ********
           * REGISTER USE
                                  *
             GLOBAL VARIABLE
              R13: LOG CPU #
              R14: DBR
                                  *
             LOCAL VARIABLES:
              R2: CURRENT_VP
              R3: TEMP VAR
                                  *
              R4: VPT.LOCK ADDR
           * R5: TEMP
           *********
           ENTRY
            ! GET LOGICAL CPU # !
0 15C 5F00
                  GET_CPU_NO !RETURNS:
            CALL
            ! LOAD IDLE DBR ON CURRENT VP !
0174 6103
           LD
                      R3. PRDS.IDLE VP
0176 OA10 .
0178 6135
                      R5. VPT. VP. DBR (R3)
           LD
017A 0010'
017C 6F25
           LD
                     VPT. VP. DBR (R2), R5
017E 0010'
            ! TURN ON CURRENT VP'S IDLE FLAG!
0180 4D25
           LD
                      VPT. VP. IDLE FLAG (R2) . #ON
0182 00161
0184 FFFF
            ! SET VP TO READY STATE !
0186 4D25
           LD
                     VPT. VP. STATE (R2), #READY
0188 00141
018A 0001
            ! SCHEDULE FIRST ELIGIBLE READY VP !
            CALL GETWORK !R13:LOGICAL CPU #
018C 5F00
018E 0000 1
                            R14:DBR!
            ! UNLOCK VPT !
0190 4D08
            CLR VPT.LOCK
0192 00001
            ! UNMASK VECTORED INTERRUPTS !
0194 7C05
           EI VI
0196 9E08
           RET
0198
          END IDLE
```

```
0198
            SWAP VDBR
                             PROCEDURE
           · ********************
            * LOADS NEW DBR ON
            * CURRENT VP. CALLED BY *
            * TC GETWORK.
            ***************
            * REGISTER USE
            *
               PARAMETERS
                R1: NEW_DBR (INPUT)
               GLOBAL VARIABLES
                R13: LOGICAL CPU #
            *
                R14: DBR
               LOCAL VARIABLES
                                     *
                R2: CURRENT VP
                R4: VPT.LOCK ADDR
                                     *
            **********
            ENTRY
             ! SAVE NEW DBR !
0198 93F1
             PUSH
                        aR15, R1
             ! MASK INTERRUPTS !
019A 7C01
             DI
                   VI
             ! LOCK VPT !
019C 7604
             LDA
                        R4, VPT.LOCK
019E 0000'
01A0 5F00
             CALL
                        SPIN_LOCK ! (R4: - VPT. LOCK) !
01A2 02821
             ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP.!
             ! GET CPU # !
01A4 5F00
             CALL
                        GET_CPU_NO
                                    ! RETURNS:
01A6 02C8
                                     R1: CPU #
                                     R2: # VP S!
01A8 A11D
                        R13, R1
             LD
             ! GET CURRENT VP !
01AA 61D2
             LD
                        R2, VPT.RUNNING_LIST (R13)
01AC 0002'
                        ! * * * DEBUG * * * !
01AE 4D21
                        CP VPT. VP. MSG LIST (R2), #NIL
01B0 001E'
01B2 FFFF
                        IF NE ! MSG WAITING !
                                                THEN
01B4 5E06
01B6 01C4
01B8 2100
                        LD
                            RO, #S_N_A ! SWAP NOT ALLOWED !
01BA 0005
                         LDA R1, $ !PC!
01BC 7601
01BE 01BC'
01C0 5F00
                        CALL MONITOR
01C2 A900
                        FI
                        ! * * END DEBUG * * !
             ! SET DBR !
01C4 612E
                        R14, VPT.VP.DBR(R2)
             LD
01C6 0010
             ! RESTORE NEW DBR !
```

```
01C8 97F0 POP RO, @R15
01CA 5F00 CALL GET_DBR_ADDR ! (RO: DBR #)
01CC 000A
                                        RETURNS
                                        (R1: DBR ADDR) !
            ! LOAD NEW DBR ON CURRENT VP !
01CE 6F21
            LD
                      VPT. VP. DBR (R2) , R1
01D0 0010 ·
            ! TURN OFF IDLE FLAG !
01D2 4D25
            LD VPT.VP.IDLE_FLAG(R2), #OFF
01D4 0016 ·
01D6 0000
            ! SET VP TO READY STATE !
01D8 4D25
            LD VPT. VP. STATE (R2), #READY
01DA 0014'
01DC 0001
            ! SCHEDULE FIRST ELGIBLE READY VP !
01DE 5F00
          CALL GETWORK !R13:LOGICAL CPU #
01E0 0000'
                             R14:DBR !
            ! UNLOCK VPT !
01E2 4D08
            CLR VPT.LOCK
01E4 0000 ·
            ! UNMASK VECTORED INTERRUPTS !
01E6 7C05
            EI VI
01E8 9E08
           RET
01EA
        END SWAP_VDBR
```

```
PHYS_PREEMPT_HANDLER PROCEDURE
01EA
           · ********************************
            * HARDWARE PREEMPT INTERRUPT
            * HANDLER. ALSO TESTS FOR
            * VIRTUAL PREEMPT INTERRUPT
            * FLAG AND INVOKES INTERRUPT
            * HANDLER IF FLAG IS SET.
            * INVOKED UPON EVERY EXIT FROM
            * KERNEL. KERNEL FCW MASKS
            * NVI INTERBUPTS TO PREVENT
            * SIMULTANEOUS PREEMPT INTERR.
            * HANDLING.
            ***********
            * REGISTER USE
               LOCAL VARIABLES
                R1: PREEMPT_INT_FLAG
                R2: CURRENT_VP
            * GLOBAL VARIABLES
                R13:LOGICAL CPU #
                R14: DBR
            ************
            ENTRY
             ! * * PREEMPT_HANDLER * * !
             ! SAVE ALL REGISTERS !
                      R15, #32
             SUB
01EA 030F
01EC 0020
                      DR15, R1, #16
01EE 1CF9
             LDM
01F0 010F
             ! SAVE NORMAL STACK POINTER (NSP) !
01F2 7D67
             LDCTL
                      R6. NSP
                      aR15, R6
01F4 93F6
             PUSH
             ! GET CPU # !
                     GET_CPU_NO ! RETURNS:
01F6 5F00
             CALL
01F8 02C8
                                  R1: CPU #
                                  R2:# VP'S!
                     R13, R1
01PA A11D
             LD
              ! MASK INTERRUPTS !
                   VI
01FC 7C01
             DI
              ! LOCK VPT !
                  R4, VPT.LOCK
01FE 7604
             LDA
0200 0000
             CALL SPIN_LOCK
0202 5F00
0204 02821
              !RETURNS WHEN VPT IS LOCKED!
              ! SET DBR !
                       R2, VPT.RUNNING_LIST(R13)
             LD
0206 61D2
0208 00021
                       R14, VPT. VP.DBR (R2)
             LD
020A 612E
020C 0010'
```

```
! PUT CURRENT PROCESS IN READY STATE !
             LD
020E 4D25
                      VPT. VP. STATE (R2) . #READY
0210 00141
0212 0001
             CALL GETWORK !R13:LOG CPU #
0214 5F00
0216 00001
                                R14: DBR 1
           PREEMPT RET:
             ! UNLOCK VPT !
0218 4D08
             CLR
                   VPT.LOCK
021A 0000'
             ! UNMASK VECTORED INTERRUPTS !
021C 7C05
             EI
                   VI
           KERNEL_EXIT:
             ! *** UNMASK VIRTUAL PREEMPTS *** !
             ! ** NOTE: SAFE SEQUENCE AND DOES NOT REQUIRE
                        VPT TO BE LOCKED. **!
             ! GET CURRENT VP !
021E 610D
             LD
                   R13, PRDS.LOG_CPU_ID
0220 0A0C*
0222 61D2
             LD R2, VPT.RUNNING LIST (R13)
0224 00021
             ! TEST PREEMPT INTERRUPT FLAG !
0226 4D21
             CP
                    VPT. VP. PREEMPT (R2), #ON
0228 0018
022A FFFF
022C 5E0E
             IF EQ ! PREEMPT FLAG IS ON! THEN
022E 0240*
                ! RESET PREEMPT FLAG !
0230 4D25
                LD
                    VPT.VP.PREEMPT(R2), #OFF
0232 00181
0234 0000
                ! SIMULATE VIRTUAL PREEMPT INTERRUPT !
0236 2101
                LD
                     R1, #0
0238 0000
023A 6112
                LD
                     R2, VPT. VIRT_INT_VEC (R1)
023C 000C'
023E 1E28
                JP
                    @R2
           !NOTE: THIS JUMP TO TRAFFIC CONTROL
            IS USED ONLY IN THE CASE OF A PREEMPT INTERRUPT.
            AND SIMULATES A HARDWARE INTERRUPT. **!
            ! *** END VIRTUAL PREEMPT HANDLER *** !
            FI
           ! NOTE: SINCE A HOWE INTERRUPT DOES NOT EXIT
               THROUGH THE GATE, THOSE FUNCTIONS PROVIDED
```

PROVIDED HERE ALSO. !

BY A GATE EXIT TO HANDLE PREEMPTS MUST BE

```
024E
           RUNNING_VP
                                   PROCEDURE
          · ************************
           * CALLED BY TRAFFIC CONTROL.
           * RETURNS VP_ID. RESULT IS VALID*
           * ONLY WHILE APT IS LOCKED.
           ***********
           * REGISTER USE
             PARAMETERS
              R1: EXT VP ID (RETURNED)
               R3: LOG CPU # (RETURNED)
             LOCAL VARIABLES
               R2: VP INDEX
           ***********
           ENTRY
            ! MASK INTERRUPTS !
024E 7C01
            DI
                 VΙ
            ! LOCK VPT !
0250 7604
            LDA
                     R4. VPT. LOCK
0252 00001
            CALL
0254 5F00
                      SPIN_LOCK ! (R4:¬VPT.LOCK) !
0256 02821
            ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
            ! GET LOGICAL CPU # !
0258 5F00
            CALL
                      GET_CPU_NO !RETURNS:
025A 02C8*
                                   R1: CPU #
                                   R2: # VP S!
025C A113
                      R3, R1
            LD
025E 6132
            LD
                      R2, VPT.RUNNING LIST (R3)
0260 00021
            ! CONVERT VP INDEX TO VP ID !
0262 6121
                      R1, VPT. VP.EXT ID (R2)
            LD
0264 00201
                      ! * * * DEBUG * * * !
0266 0B01
                      CP R1, #NIL
0268 FFFF
026A 5E0E
                     IF EQ ! KERNEL PROC! THEN
026C 027A'
026E 2100
                      LD RO, #V_I_E ! VP INDEX ERROR!
0270 0006
0272 7601
                      LDA R1. $
0274 02721
0276 5F00
                      CALL MONITOR
0278 A900
                      FI
                      ! * * END DEBUG * * !
            ! UNLOCK VPT !
027A 4D08
            CLR
                     VPT LOCK
027C 0000 ·
            ! UNMASK VECTORED INTERRUPTS !
027E 7C05
           EI
                 VΙ
0280 9E08
           RET
0282
         END RUNNING VP
```

```
0282
              SPIN LOCK PROCEDURE
              · **********************
              * USES SPIN_LOCK MECH.
              * LOCKS UNLOCKED DATA
              * STRUCTURE (POINTED TO *
              * BY INPUT PARAMETER). *
              *********
              *REGISTER USE
               * PARAMETERS
                R4: LOCK ADDR (INPUT) *
              **************
               ENTRY
                NOTE: SINCE ONLY ONE PROCESSOR CURRENTLY
                   IN SYSTEM, LOCK NOT NECESSARY. **!
! * * * DEBUG * * *!
0282 0D41
                 aR4, #OFF
              CP
0284 0000
0286 5E06
              IF NE ! NOT UNLOCKED ! THEN
0288 0296
028A 2100
               LD RO, #U_L ! UNAUTHORIZED LOCK!
028C 0000
028E 7601
               LDA R1, $
0290 028E
0292 5F00
               CALL MONITOR
0294 A900
                        ! * * END DEBUG * * !
                      TEST_LOCK:
                       ! DO WHILE STRUCTURE LOCKED !
                       DR4
0296 OD46
             TSET
             JR MI, TEST_LOCK
0298 E5FE
                        ! ** NOTE SEE PLZ/ASM MANUAL
                                  FOR RESTRICTIONS ON
                                  USE OF TSET. **!
029A 9E08
           RET
029C
         END SPIN_LOCK
```

```
ITC_GET_SEG_PTR
029C
                                   PROCEDURE
           *****************
           * GETS BASE ADDRESS OF SEGMENT
           * INDICATED.
           ***********
           * REGISTER USE:
              RO:SEG BASE ADDRESS (RET)
              R1:SEG NR (INPUT)
             R2: RUNNING VP (LOCAL)
            * R3:DBR_VALUE (LOCAL)
             R4: VPT.LOCK
             R13:LOGICAL CPU #
           ************
           ENTRY
            ! SAVE SEGMENT # !
029C 93F1
            PUSH OR 15, R1
            ! MASK INTERRUPTS !
029E 7C01
            DI
                 VI
            ! LOCK VPT !
02A0 7604
            LDA
                   R4. VPT.LOCK
02A2 0000*
02A4 5F00
            CALL SPIN LOCK ! R4: - VPT.LOCK!
02A6 0282 *
            ! GET CPU # !
02A8 5F00
            CALL
                  GET_CPU_NO
                               !RETURNS:
02AA 02C8*
                                 R1: CPU #
                                 R2:# VP'S!
02AC A11D
            LD
                    R13, R1
            ! RESTORE SEGMENT # !
02AE 97F1
            POP
                   R1, @R15
02B0 61D2
            LD
                    R2, VPT.RUNNING LIST (R13)
02B2 0002°
02B4 6123
            LD
                    R3, VPT. VP. DBR (R2)
02B6 00101
            ! UNLOCK VPT !
02B8 4D08
            CLR
                   VPT.LOCK
02BA 0000°
            ! UNMASK VECTORED INTERRUPTS !
02BC 7C05
            EI
                  VI
02BE 1900
                  RRO.#4
            MULT
02C0 0004
02C2 7130
                   RO, R3 (R1)
            LD
02C4 0100
02C6 9E08
            RET
02C8
          END ITC_GET_SEG_PTR
```

```
02C8
          GET_CPU_NO PROCEDURE
          · *********************
          * FIND CURRENT CPU NO
          * CALLED BY DIST MMGR
          * AND MM
          *********
          * RETURNS
          * R1: CPU_NO
                                *
          * R2: # OF VP'S
          **********
          ENTRY
02C8 6101
           LD
                   R1, PRDS.LOG_CPU_ID
02CA OAOC
02CC 6102
           LD
                  R2, PRDS. VP_NR
O2CE OAOE
02D0 9E08
           RET
02D2
        END GET_CPU_NO
02D2
        K LOCK
                          PROCEDURE
          · **********************
             STUB FOR WAIT LOCK
          **********
            R4: -LOCK (INPUT)
          **********
          ENTRY
02D2 5F00
           CALL SPIN_LOCK
02D4 02821
02D6 9E08
           RET
02D8
       END K LOCK
02D8
        K UNLOCK
                          PROCEDURE
          · ********************
             STUB FOR WAIT UNLOCK *
          **********
          * R4:¬LOCK (INPUT)
          **********
          ENTRY
           CLR
02D8 0D48
                3 R4
           RET
02DA 9E08
02DC
       END K_UNLOCK
       END INNER_TRAFFIC_CONTROL
```

Appendix H

SEGMENT MANAGER LISTINGS

```
Z8000ASM 2.02
LOC OBJ CODE
                   STMT SOURCE STATEMENT
        $LISTON $TTY
                         MODULE
        SEG_MGR
        CONSTANT
          NULL_SEG
                               := -1
          NULL ACCESS
                               := 4
          MAX_SEG_NO
                               := 64
          MAX_NO_KST_ENTRIES := 54
          MAX_SEG_SIZE
                               := 128
          KST_SEG_NO
                               := 2
          NR_OF_KSEGS
                            := 10
          TRUE
                                := 1
          FALSE
                                := 0
                                := 1
          READ
          WRITE
                                := 0
        ! **** SUCCESS_CODES
                                  **** !
                                := 2
          SUCCEEDED
          MENTOR_SEG_NOT_KNOWN := 22
          ACCESS_CLASS_NOT_EQ := 33
          NOT_COMPATIBLE
                               := 24
          SEGMENT_TOO_LARGE
                               := 25
                               := 27
          NO_SEG_AVAIL
          SEGMENT_NOT_KNOWN
                              := 28
          SEGMENT_IN_CORE
                               := 29
          KERNEL_SEGMENT
                               := 30
                               := 31
          INVALID_SEGMENT_NO
          NO ACCESS PERMITTED
                               := 32
         LEAF_SEG_EXISTS
NO_LEAF_EXISTS
                               := 10
                               := 11
          ALIAS_DOES_NOT_EXIST := 23
          NO_CHILD_TO_DELETE
                               := 20
         G_AST_FULL
L_AST_FULL
                                := 12
                               := 13
          PROC_CLASS_NOT_GE_SEG_CLASS := 41
          LOCAL_MEMORY_FULL := 16
          GLOBAL_MEMORY_FULL
                              := 17
```

SEC_STOR_FULL

:= 21

```
MONITOR
                      := %059A
TYPE
  H_ARRAY
             ARRAY [ 3 WORD ]
  KST_REC RECORD
   [ MM_HANDLE
                  H_ARRAY
    SIZE
                   WORD
    ACCESS MODE
                   BYTE
    IN_CORE
                   BYTE
    CLASS
                   LONG
    M_SEG_NO
                   SHORT_INTEGER
   M_SEG_NO SHORT_INTEGER ENTRY_NUMBER SHORT_INTEGER
  ADDRESS
             WORD
  SEG_ARRAY ARRAY [MAX_SEG_SIZE BYTE]
INTERNAL
$SECTION SM_KST_DCL
! NOTE: THIS SECTION IS AN "OVERLAY"
 OR "FRAME" USED TO DEFINE THE
 FORMAT OF THE KST. NO STORAGE
 IS ASSIGNED BUT RATHER THE KST IS
  STORED IN A SEPARATELY OBTAINED
 AREA (A SEGMENT SET ASIDE FOR IT) !
$ABS 0
KST
      ARRAY MAX_NO_KST_ENTRIES KST_REC
EXTERNAL
  CLASS EQ PROCEDURE
  CLASS GE PROCEDURE
  MM_CREATE_ENTRY PROCEDURE
  MM_DELETE_ENTRY PROCEDURE
  MM ACTIVATE PROCEDURE
  MM DEACTIVATE PROCEDURE
  MM_SWAP_IN PROCEDURE
  MM_SWAP_OUT PROCEDURE
 PROCESS_CLASS PROCEDURE
 ITC GET SEG PTR PROCEDURE
  GET DBR NUMBER PROCEDURE
```

0000

\$SECTION SM_PROC GLOBAL

```
0000
                             PROCEDURE
          CREATE_SEG
         ! CHECKS VALIDITY OF CREATE
         ! REQUEST AND
         ! CALLS MM CREATE IF VALID.
         ! REGISTER USE:
         1
            PARAMETERS
                                       :
            R1: MENTOR_SEG_NO (INPUT)
            R2: ENTRY NO (INPUT)
         1
            R3: SIZE (INPUT)
         1
            RR4: CLASS (INPUT)
            RO: SUCCESS_CODE (RETURNED)
            LOCAL USE
         į
            R9: KST REC INDEX
         1
            R6.R7 VARIOUS USES
            R13: ¬KST
         ENTRY
0000 0B03
           CP R3, #MAX_SEG_SIZE
0002 0080
0004 5E02
           IF GT THEN
0006 0010
0008 2100
             LD
                RO, #SEGMENT_TOO_LARGE
000A 0019
000C 5E08
           ELSE
000E 0CA2
0010 030F
             SUB
                   R15,#10
                            ISTACK AREA FOR
                              INPUT REGS!
0012 000A
0014 1CF9
             LDM
                   @R15,R1,#5
0016 0104
0018 2101
             LD
                   R1, #KST_SEG_NO
001A 0002
001C 5F00
             CALL
                   ITC_GET_SEG_PTR
                                  !R1: KST_SEG_NO!
001E 0000*
                    !RET:RO:¬KST!
0020 A10D
                   R13, RO !KST BASE ADDRESS
             LD
                            (IE ¬KST)!
0022 1CF1
             LDM
                   R1. DR15. #5 ! RESTORE NEEDED REGS!
0024 0104
0026 A119
             LD
                   R9,R1 !COPY OF MENTOR SEG NO!
0028 0309
             SUB
                   R9, #NR OF KSEGS
                                   !CONVERT
                                    MENTOR_SEG_NO
002A 000A
                                    KST_REC INDEX!
002C 1908
             MULT RR8, #SIZEOF KST REC
                               !OFFSET TO KST_REC!
```

```
002E 0010
0030 819D
              ADD R13, R9 ! ADD OFFSET TO KST
                                   BASE ADDRESS!
             LD R6, #NULL_SEG
0032 2106
0034 FFFF
0036 4ADE
             CPB
                   RL6, KST.M_SEG NO (R13)
0038 000E
003A 5E0E
             IF EQ THEN ! MENTOR SEG NOT KNOWN!
003C 0046
003E 2100
                LD
                      RO, #MENTOR_SEG_NOT_KNOWN
0040 0016
0042 5E08
             ELSE
0044 009E
0046 93FD
                PUSH
                      DR15, R13
0048 5F00
                CALL PROCESS_CLASS ! RR2: PROC_CLASS!
004A 0000*
004C 97FD
               POP
                      R13, @R15
004E 54D4
               LDL
                      RR4, KST. CLASS (R13)
0050 000A
0052 93FD
                PUSH DR15,R13
0054 5F00
                CALL CLASS_EQ !RR2: PROC_CLASS!
0056 0000*
                           !RR4: MENTOR SEG CLASS!
                           !R1: (RET) CONDITION CODE!
0058 97FD
                      R13, @R15
               POP
005A A116
               LD
                      R6, R1
005C 1CF1
             LDM
                      R1, aR15, #5 !RESTORE INPUT REGS!
005E 0104
0060 OB06
               CP R6.#FALSE
0062 0000
0064 5E0E
               IF EO THEN
0066 0070
0068 2100
                  LD
                        RO, #ACCESS_CLASS_NOT_EQ
006A 0021
006C 5E08
               ELSE
006E 009E
                        DR 15. R13
                                 !SAVE ¬KST!
0070 93FD
                  PUSH
                        RR2,RR4
                                  !CLASS!
0072 9442
                  LDL
                       RR4, KST. CLASS (R13)
0074 54D4
                  LDL
0076 000A
0078 5F00
                  CALL CLASS_GE !RR2:CLASS!
007A 0000*
                           !RR4: MENTOR CLASS!
                          !RET:R1:COND_CODE!
                        R13, aR15 ! RESTORE PTR!
                 20 B
007C 97FD
                  CP
                        R1. #FALSE
007E 0B01
0080 0000
                 LDM R1, @R15, #5
0082 1CF1
0084 0104
0086 SEOE
                 IF EO THEN
0088 0092
                          RO, #NOT_COMPATIBLE
                    LD
008A 2100
```

```
008C 0018
008E 5E08
                  ELSE
0090 009E*
0092 76D1
                    LDA
                           R1, KST. MM_HANDLE (R13)
0094 0000
0096 5F00
                     CALL MM_CREATE_ENTRY
0098 0000*
                     !R1:PTR TO MM_HANDLE!
                     !R2:ENTRY_NO!
                     !R3:SIZE!
                     !RR4:CLASS!
                     !RO: (RETURNED) SUCCESS_CODE!
009A 5F00
                     CALL CONFINEMENT_CHECK
009C 04281
                       ! (RO:SUCCESS_CODE)!
                   FI
                FI
              FI
009E 010F
                    R15,#10
              A DD
00A0 000A
            FI
00A2 9E08
            RET
00A4
           END CREATE_SEG
```

```
DELETE_SEG PROCEDURE
00A4
         ! CHECKS VALIDITY OF DELETE
         ! REQUEST AND
            CALLS MM_DELETE IF VALID.
         REGISTER USE:
            PARAMETERS
             R1: MENTOR_SEG_NO (INPUT)
         1
         !
             R2: ENTRY_NO (INPUT)
             RO: SUCCESS_CODE (RET)
            LOCAL USE
             R6: VARIOUS LOCAL USES
         ENTRY
00A4 93F1
           PUSH
                 DR15,R1
                          !SAVE NEEDED REGS!
00A6 93F2
           PUSH
                 aR15, R2
                 R1, #KST_SEG_NO
00A8 2101
           LD
00AA 0002
00AC 5F00
           CALL
                 ITC_GET_SEG_PTR    !R1: KST_SEG_NO!
00AE 0000*
00B0 A10D
                 R13,R0
                         !¬KST!
          LD
00B2 97F2
          POP
                 R2, DR 15
                         !RESTORE INPUT REGS!
00B4 97F1
          POP
                 R1, @R15
                 R1, #NR_OF_KSEGS
                                  !CONVERT
00B6 0301
           SUB
                               MENTOR_SEG_NO TO
00B8 000A
                                  KST REC INDEX!
00BA 1900
                 RRO, #SIZEOF KST REC
                                    !OFFSET
          MULT
                                 TO DESIRED REC!
00BC 0010
                 R13,R1 !ADD OFFSET TO KST BASE
00BE 811D
           A DD
                                        ADDRESS!
00C0 2106
           LD
                 R6, #NULL_SEG
00C2 FFFF
                 RL6, KST. M_SEG_NO(R13)
OOC4 4ADE
           CPB
00C6 000E
                           !MENTOR SEGMENT
00C8 5E0E
           IF
                 EO
                      THEN
                                NOT KNOWN!
00CA 00D4 *
00CC 2100
             LD
                   RO, #MENTOR_SEG_NOT_KNOWN
00CE 0016
00D0 5E08
           ELSE
00D2 010E'
                           !SAVE NEEDED REGS!
                   DR15.R1
00D4 93F1
             PUSH
                   @R15,R2
00D6 93F2
             PUSH
                   DR15, R13
00D8 93FD
             PUSH
                  PROCESS_CLASS
00DA 5F00
             CALL
00DC 0000*
                   ! (RETURNS RR2:PROC_CLASS) !
                   R13, DR15
             POP
00DE 97FD
                   RR4, KST.CLASS (R13) !MENTOR
00E0 54D4
             LDL
```

SEG CLASS!

```
00E2 000A
00E4 93FD
              PUSH @R15,R13
00E6 5F00
              CALL CLASS_EQ !RR2:PROCESS CLASS!
00E8 0000*
                          !RR4:MENTOR SEG CLASS!
                          !R1: (RET) CONDITION_CODE!
00EA A116
              LD
                     R6, R1
00EC 97FD
              POP
                     R13, DR15
00EE 97F2
                     R2, aR15 ! RESTORE NEEDED REGS!
              POP
00F0 97F1
              POP
                     R1, @R15
00F2 0B06
              CP
                     R6. #FALSE
00F4 0000
00F6 5E0E
            IF EO
                         THEN
00F8 0102 ·
00FA 2100
                LD
                      RO, #ACCESS_CLASS_NOT_EQ
00FC 0021
00FE 5E08
             ELSE
0100 010E*
0102 76D1
                LDA
                      R1, KST. MM_HANDLE (R13)
0104 0000
0106 5F00
                CALL MM_DELETE_ENTRY
0108 0000*
              !R1: -MM_HANDLE!
              ! R2: ENTRY_NO!
              !RO: (RET) SUCCESS_CODE!
010A 5P00
                     CALL CONFINEMENT_CHECK
010C 0428
              ! (RO:SUCCESS_CODE)!
              FI
            FI
010E 9E08
            RET
0110
           END DELETE_SEG
```

```
0110
           MAKE KNOWN
                               PROCEDURE
          ! CHECKS VALIDITY OF MAKE KNOWN !
          ! REQUEST AND CALLS MM_ACTIVATE !
          ! IF VALID. ASSIGNS SEG
          ! NUMBER AND UPDATES KST.
          ! REGISTER USE:
          ! PARAMETERS:
             R1: MENTOR_SEG_NO (INPUT)
          .
             R2:ENTRY_NO(INPUT)
             R3:ACCESS DESIRED (INPUT)
             RO:SUCCESS_CODE (RET)
             R1:SEGMENT_NO (RET)
             R2:ACCESS_ALLOWED (RET)
          ! LOCAL USE
             IDENTIFIED AT POINT OF USAGE !
          ENTRY
0110 93F1
            PUSH
                 aR 15, R1
                           !SAVE INPUT REGS!
0112 91F2
            PUSHL @R15,RR2
0114 2101
           LD
                 R1, #KST_SEG_NO
0116 0002
0118 5F00
           CALL
                 ITC_GET_SEG_PTR ! (R1: KST_SEG_NO,
                                    RET:RO: ¬KST) !
011A 0000*
011C A10D
                  R13,R0
            LD
                          !¬KST!
011E 95F2
           POPL
                  RR2, aR15
0120 97F1
           POP
                  R1, DR 15
0122 A115
           LD
                  R5,R1 !COPY OF MENTOR_SEG_NO!
                  R5, #NR OF KSEGS ! CONVERT TO
0124 0305
            SUB
                                        INDEX!
0126 000A
0128 1904
            MULT
                  RR4, #SIZEOF KST_REC !KST OFFSET
                                      TO SEG REC!
012A 0010
                          !ADD OFFSET TO -KST!
012C 815D
                  R13.R5
            A DD
012E 2104
                  R4, #NULL_SEG
           LD
0130 FFFF
                  RL4, KST. M_SEG_NO(R13)
0132 4ADC
            CPB
0134 000E
0136 5E0E
            IF EQ THEN
0138 014A'
013A 2100
                    RO, #MENTOR_SEG_NOT_KNOWN
             LD
013C 0016
013E 2101
                    R1, #NULL_SEG
             LD
0140 FFFF
                    R2, #NULL_ACCESS
0142 2102
             LD
0144 0004
0146 5E08
           ELSE
0148 0208
014A 2107
             LD
                    R7,#0 !KST INDEX!
014C 0000
```

```
014E 2108
                   R8, #NULL_SEG ! AVAIL SEG INDEX!
             LD
0150 FFFF
                    R9,R0 !¬KST!
0152 A109
              LD
0154 210A
              LD
                   R10, #NULL_SEG !SEG KNOWN INDICATOR!
0156 FFFF
            SEE IF KNOWN:
            DO
0158 4A99
                    RL1, KST.M_SEG_NO (R9)
              CPB
015A 000E
015C 5E0E
             IF EQ THEN
015E 017C*
0160 4A9A
                CPB RL2, KST. ENTRY NUMBER (R9)
0162 000F
0164 5E0E
                IF EQ THEN !CASE: SEG KNOWN!
0166 017C*
0168 2100
                  LD
                       RO.#SUCCEEDED
016A 0002
016C 0107
                  ADD R7, #NR_OF_KSEGS
016E 000A
0170 A171
                  LD
                        R1, R7 !SEG#!
0172 609A
                  LDB
                        RL2, KST. ACCESS_MODE (R9)
0174 0008
0176 A11A
                  LD
                        R10,R1 !SET SEG KNOWN
                                      INDICATOR!
0178 5E08
                  EXIT FROM SEE_IF_KNOWN
017A 01A6
                PI
              FI
017C 4A9C
                    RL4, KST.M_SEG_NO(R9)
              CPB
                    !SEE IF SEG # AVAIL!
017E 000E
              IF EQ THEN
0180 5E0E
0182 0192
0184 0B08
                CP R8, #NULL_SEG
0186 FFFF
0188 5E0E
                IF EQ THEN
018A 0192'
018C A178
                  LD R8,R7 !SAVE FIRST
                                AVAIL SEG INDEX!
018E 0108
                  ADD
                         R8, #NR_OF_KSEGS
                              !CONVERT TO SEG #!
0190 000A
                FI
              FI
0192 A970
              INC
                    R7
0194 0109
                    R9, #SIZEOF KST_REC
              A DD
                              !INCREMENT ONE REC!
0196 0010
0198 0B07
              CP
                    R7, #MAX_NO_KST_ENTRIES
019A 0036
019C 5E02
              IF GT THEN
019E 01A4*
```

```
01A0 5E08
                  EXIT FROM SEE_IF_KNOWN
01A2 01A6 1
               FI
01A4 E8D9
             OD
             !SEE_IF_KNOWN!
01A6 0B0A
                   R10, #NULL_SEG
01A8 FFFF
Olaa SEOE
               IF EQ THEN
                            ISEG KNOWN
                             INDICATOR NOT SET!
01AC 02C8
01AE 0B08
                 CP
                        R8, #NULL SEG
01BO FFFF
01B2 5E06
                 IF NE THEN
                              !CASE:SEG UNKNOWN
                                 AND SEG# AVAIL!
01B4 02BC*
01B6 91F0
                   PUSHL DR 15, RRO
                                     !¬KST AND
                                       MENTOR_SEG_NO!
01B8 91F2
                   PUSHL DR15, RR2 !ENTRY_NO
                                     &ACCESS_DESIRED!
01BA 93F8
                   PUSH
                          DR15, R8 ! AVAIL SEG
                                     INDEX IN KST!
01BC 93FD
                   PUSH
                          DR 15, R13 !MENTOR SEG REC PTR!
01BE 5F00
                   CALL
                          GET_DBR_NUMBER
                   ! (RET:RL1:DBR NO)!
01C0 0000*
01C2 A11A
                          R10,R1 !DBR_NO!
                   LD
01C4 97FD
                   POP
                          R13, @R15
01C6 97F8
                          R8, DR 15
                   POP
01C8 95F2
                   POPL
                          RR2, aR15
01CA 95F0
                          RRO, DR15
                   POPL
             !MUST REARRANGE REGS FOR PASSING AND
              RETURN CONSISTENCY OF LOCATION!
01CC A135
               000D
047C 5E0E
                   LD
                          R5,R3
                                 !ACCESS_DESIRED!
                                 !ENTRY_NO!
01CE A123
                   LD
                          R3, R2
01D0 76D2
                   LDA
                          R2, KST. MM_HANDLE (R13) !HPTR!
01D2 0000
                          R6 . R1
                                 !MENTOR_SEG_NO!
01D4 A116
                   LD
01D6 A181
                   LD
                          R1, R8
                                !SEGMENT_NO (SAVE)!
01D8 A184
                   LD
                          R4,R8
                                !SEGMENT_NO
                                 (PASSING ARG)!
                          R9,R0
01DA A109
                   LD
                                  !¬KST!
01DC 030F
                   SUB
                          R15,#20
01DE 0014
                          DR 15, R1, #10 !SAVE REGS 1-10!
01E0 1CF9
                   LDM
01E2 0109
                                   !DBR_NO PASSED
01E4 A1A1
                   LD
                          R1.R10
                                    IN R1!
                   LD
                          R11, R8
01E6 A18B
01E8 030B
                   SUB
                          R11, #NR_OF_KSEGS
01EA 000A
                          RR 10, #SIZEOF KST REC
01EC 190A
                   MULT
01EE 0010
```

```
01FO A1BC
                   LD
                          R12, R11
01F2 819C
                   ADD
                          R12, R9
                          MM_ACTIVATE
01F4 5F00
                   CALL
01F6 0000*
            ! (R1:DBR_NO, R2: HPTR, R3:ENTRY_NO,
              R4:SEGMENT NO, R12:RET_HPTR)!
            ! (RET: RO: SUCCESS_CODE, RR2:CLASS,
              R4:SIZE)!
01F8 5F00
                   CALL CONFINEMENT CHECK
                        ! (RO:SUCCESS_CODE) !
01FA 0428
01FC 942A
                   LDL
                          RR10.RR2 !CLASS!
01FE A14C
                   LD
                          R12,R4 !SIZE!
0200 1CF1
                   LDM
                          R1, aR15, #9 ! RESTORE REGS 1-9!
0202 0108
0204 A187
                          R7, R8 !SEG #!
                   LD
0206 0307
                   SUB
                          R7, #NR_OF_KSEGS
0208 000A
020A 1906
                  MULT
                          RR6, #SIZEOF KST REC
                              !OFFSET TO REC!
020C 0010
020E A17D
                          R13, R7
                   LD
0210 819D
                   ADD
                          R13, R9 !ADD ¬KST TO OFFSET!
0212 5DDA
                  LDL
                          KST.CLASS (R13), RR10 !CLASS!
0214 000A
0216 6FDC
                  LD
                          KST.SIZE(R13),R12 !SIZE!
0218 0006
021A 0A08
                  CPB
                         RLO, #SUCCEEDED
021C 0202
021E 5E0E
                  IF EQ THEN
0220 02AC1
0222 93FD
                     PUSH
                            DR15, R13
0224 5F00
                     CALL PROCESS CLASS
0226 0000*
                     ! (RET: RR2: PROC_CLASS) !
0228 97FD
                     POP
                            R13,0R15
022A 54D4
                     LDL
                            RR4, KST.CLASS (R13)
022C 000A
022E 93FD
                     PUSH
                            DR15, R13
0230 91F2
                     PUSHL @R15,RR2
0232 91F4
                     PUSHL @R15,RR4
0234 5F00
                     CALL CLASS_GE
0236 0000*
               ! (RR2: PROC_CLASS, RR4: SEG CLASS, RET:
                      R1:CONDITION_CODE) !
0238 95F4
                     POPL RR4, aR15
023A 95F2
                            RR2, DR15
                     POPL
023C 97FD
                            R13,0R15
                     POP
023E 0B01
                     CP
                           R1, #FALSE
0240 0000
0242 5E0E
                    IF EQ THEN !NO ACCESS
                                  POSSIBLE--DEACT.!
0244 02661
```

```
0246 1CF1
                       LDM
                             R1, DR 15, #10
0248 0109
024A A1A1
                       LD
                              R1,R10 !DBR NO!
024C 76D2
                       LDA
                             R2, KST. MM_HANDLE (R13)
                                         !HPTR!
024E 0000
0250 5F00
                       CALL MM_DEACTIVATE
                           !RET:RO:S_CODE!
0252 0000*
0254 5F00
                       CALL CONFINEMENT CHECK
                                      !RO:S_CODE!
0256 0428
0258 21F1
                       LD
                              R1, @R15 !SEG #!
025A 2102
                             R2, #NULL ACCESS
                       LD
025C 0004
025E 2100
                       LD RO.
                         #PROC_CLASS_NOT_GE_SEG_CLASS
0260 0029
0262 5E08
                    ELSE
0264 02A8*
0266 93FD
                       PUSH DR15, R13
0268 5F00
                       CALL CLASS_EQ !(RR2:PROC_CLASS,
026A 0000*
                                        RR4:SEG CLASS,
                              RET: R1: CONDITION CODE)!
                      LD RO,R1 ! CONDITION_CODE!
LDM R1. ar 15 +0
026C 97FD
026E A110
0270 1CF1
0272 0108
0274 OB00
                       CP RO, #TRUE
0276 0001
0278 SEOE
                      IF EQ THEN
027A 0290 1
                         CP R5, #WRITE
027C 0B05
027E 0000
                         IF EQ THEN
0280 5E0E
0282 028A
0284 CA00
                           LDB
                                 RL2, #WRITE
0286 5E08
                         ELSE
0288 028C4
                                 RL2, #READ
                           LDB
028A CA01
                         FI
028C 5E08
                       ELSE
028E 0292
                         LDB RL2,#READ
0290 CA01
                       PΙ
                             KST.IN_CORE (R13), #FALSE
                       LDB
0292 4CD5
0294 0009
0296 0000
                             KST.M_SEG_NO(R13),RL6
                      LDB
0298 6EDE
029A 000E
                             KST.ENTRY_NUMBER (R13), RL3
                       LDB
029C 6EDB
029E 000F
```

```
02A0 6EDA
                      LDB KST.ACCESS_MODE(R13), RL2
02A2 0008
02A4 2100
                      LD RO, #SUCCEEDED
                                    !SUCCESS_CODE!
02A6 0002
                    FI
02A8 5E08
                  ELSE
02AA 02B4*
02AC 2101
                    LD
                       R1, #NULL_SEG
02AE FFFF
02B0 2102
                   LD
                       R2, #NULL_ACCESS
02B2 0004
                  FI
                  ADD
                       R15, #20
02B4 010F
02B6 0014
02B8 5E08
                ELSE
02BA 02C8*
02BC 2100
                  LD
                       RO, #NO_SEG_AVAIL
02BE 001B
02C0 2101
                  LD
                       R1, #NULL_SEG
02C2 FFFF
02C4 2102
                  LD R2, #NULL_ACCESS
02C6 0004
                FI
              ΡI
            ΡI
02C8 9E08
           RET
02CA
           END MAKE_KNOWN
```

```
02CA
          TERMINATE
                              PROCEDURE
         ! CHECKS VALIDITY OF TERMINATE !
          ! REQUEST AND CALLS
          ! MM DEACTIVATE IF VALID
          ! REGISTER USE
           PARAMETERS
            R1:SEGMENT_NO (INPUT)
            RO:SUCCESS_CODE(RET)
          ! LOCAL USE
            R3:KST REC INDEX
            R6:CONSTANT STORAGE
            R13: ¬KST
          ENTRY
                 R3,R1 !COPY OF SEG #!
02CA A113
           LD
02CC 0303
                 R3, #NR_OF_KSEGS
            SUB
             !CONVERT SEG# TO KST INDEX!
02CE 000A
                 RR2, #SIZEOF KST_REC
02D0 1902
           MULT
02D2 0010
02D4 93F1
            PUSH
                 DR15,R1
02D6 93F3
                 DR 15, R3
           PUSH
                 R1, #KST_SEG_NO
02D8 2101
           LD
02DA 0002
                 ITC GET_SEG_PTR
02DC 5F00
           CALL
                 ! (R1: KST_SEG_NO)!
02DE 0000*
                 ! (RETURNS: RO: KST_SEG_PTR)!
02E0 A10D.
                 R13,R0
            LD
            POP
                 R3, @R15
02E2 97F3
                  R1, @R15
02E4 97F1
            POP
                 R13, R3 ! ADD OFFSET TO ¬KST!
02E6 813D
            ADD
02E8 2106
            LD
                 R6, #NULL_SEG
02EA FFFF
                 RL6.KST.M_SEG_NO(R13)
02EC 4ADE
            CPB
02EE 000E
                       THEN
02F0 5E0E
            IF
                 EQ
02F2 02FC'
                        RO, #SEGMENT_NOT_KNOWN
                  LD
02F4 2100
02F6 001C
02F8 5E08
            ELSE
02FA 0346
02FC 2106
                  LD
                        R6. #TRUE
02FE 0001
                        RL6, KST. IN_CORE (R13)
                 CPB
0300 4ADE
0302 0009
0304 SEOE
              IF
                    ΕQ
                          THEN
0306 0310
                    LD
                         RO, #SEGMENT_IN_CORE
0308 2100
```

```
030A 001D
030C 5E08
             ELSE
030E 0346
0310 0B01
                    CP R1, #NR_OF_KSEGS
0312 000A
                IF LT
0314 5E09
                             THEN
0316 0320 *
0318 2100
                  LD
                       RO, #KERNEL_SEGMENT
031A 001E
031C 5E08
               ELSE
031E 0346
0320 93FD
                  PUSH @R15,R13
0322 5F00
                  CALL GET_DBR_NUMBER
0324 0000*
                  ! (RETURNS:RL1:DBR_NO)!
0326 97FD
                  POP R13, 0R15
0328 76D2
                  LDA
                        R2, KST. MM_HANDLE (R13)
032A 0000
032C 93FD
                  PUSH @R15,R13
032E 5F00
                  CALL MM_DEACTIVATE ! (R1:DBR_NO)!
0330 0000*
                            ! (R2: -MM_HANDLE) !
                       ! (RET:RO:SUCCESS_CODE) !
0332 5F00
                  CALL CONFINEMENT_CHECK
0334 0428
            ! (RO:SUCCESS_CODE)!
0336 97FD
                  POP
                        R13,0R15
0338 0A08
                  CPB
                        RLO, #SUCCEEDED
033A 0202
033C 5E0E
                  IF EO
                               THEN !UPDATE KST!
033E 0346
0340 4CD5
                    LDB KST.M_SEG_NO(R13),
0342 000E
0344 FFFF
                                 #NULL SEG
                  FI
                FI
              FI
            FI
0346 9E08
            RET
0348
           END TERMINATE
```

```
0348
           SM_SWAP_IN
                              PROCEDURE
          ! CHECKS VALIDITY OF SWAP IN !
          ! REQUEST AND CALLS
                                        1
          ! MM_SWAP_IN IF VALID
          [********************
            REGISTER USE
          ! PARAMETERS
             R1:SEGMENT_NO (INPUT)
             RO:SUCCESS_CODE (RET)
          ! LOCAL USE
          1
             R7:KST REC INDEX
             R3:ACCESS_MODE
             R6:CONSTANT STORAGE
             R13:¬KST
          [*****************
           ENTRY
0348 A117
            LD
                  R7,R1
                          !COPY OF SEG #!
034A 0307
            SUB
                  R7, #NR_OF_KSEGS
            !CONVERT SEG# TO KST INDEX!
034C 000A
034E 1906
            MULT
                  RR6, #SIZEOF KST REC
                   !OFFSET TO KST_REC!
0350 0010
0352 93F1
                            !SAVE SEGMENT#!
            PUSH
                  DR 15, R1
0354 93F7
                  aR 15, R7
            PUSH
0356 2101
            LD
                  R1, #KST_SEG_NO
0358 0002
035A 5F00
                  ITC GET_SEG_PTR
            CALL
                                    !R1:KST SEG NO!
035C 0000*
035E A10D
                  R13,R0
                           !¬KST!
            LD
                  R7, DR 15
0360 9777
            POP
0362 97F1
            POP
                  R1, DR15 !RETRIEVE SEGMENT#!
0364 817D
            A DD
                  R13,R7 !ADD OFFSET TO KST BASE ADDR!
0366 2106
            LD
                  R6, #NULL_SEG
0368 FFFF
036A 4ADE
            CPB
                  RL6, KST. M_SEG_NO(R13)
036C 000E
036E 5E0E
            IF
                  EQ
                       THEN
0370 037A1
0372 2100
              LD
                    RO, #SEGMENT_NOT_KNOWN
0374 001C
0376 5E08
            ELSE
0378 03B81
037A 2106
              LD
                    R6. #TRUE
037C 0001
                    RL6, KST.IN_CORE (R13)
037E 4ADE
              CPB
0380 0009
              IF EQ THEN
0382 SEOE
0384 038E'
                      RO, #SUCCEEDED
0386 2100
                LD
```

```
0388 0002
038A 5E08
             ELSE
038C 03B8
038E 93FD
                PUSH
                      DR15,R13 !SAVE KST REC ADDR!
0390 5F00
                      GET_DBR_NUMBER !R1: (RET) DBR_NO!
                CALL
0392 0000*
0394 97FD
                POP
                      R13. @R15
0396 76D2
                LDA
                      R2, KST. MM_HANDLE (R13)
0398 0000
039A 60DB
                LDB
                      RL3, KST. ACCESS_MODE (R13)
039C 0008
039E 93FD
                      @R15,R13 !SAVE SEG KST REC ADDR!
                PUSH
03A0 5F00
                CALL MM_SWAP_IN !R1:DBR_NO !
03A2 0000*
                !R2: -MM_HANDLE!
                !R3: ACCESS _MODE!
                !RO: (RET) SUCCESS_CODE!
03A4 5F00
                CALL CONFINEMENT_CHECK
                      ! (RO:SUCCESS_CODE)!
03A6 0428 ·
03A8 97FD
                POP
                      R13, @R15
03AA 0A08
                CPB
                      RLO, #SUCCEEDED
03AC 0202
OBAE SEGE
                IF EQ
                             THEN
03B0 03B8 •
                 LDB KST.IN_CORE (R13), #TRUE
03B2 4CD5
03B4 0009
03B6 0101
                FI
              FI
            FI
03B8 9E08
            RET
           END SM_SWAP_IN
03BA
```

```
03BA
          SM_SWAP_OUT
                              PROCEDURE
          ! CHECKS VALIDITY OF SWAP OUT !
          ! REQUEST AND CALLS
                                       Ī
          ! MM_SWAP OUT IF VALID
          [**********************
          ! REGISTER USE
          ! PARAMETERS
            R1:SEGMENT_NO
             RO:SUCCESS_CODE (RET)
          ! LOCAL USE
            R7:KST REC INDEX
             R6:CONSTANT STORAGE
            R13: ¬KST
          [************************
          ENTRY
03BA A117
           LD
                 R7,R1 !COPY OF SEG #!
03BC 0307
            SUB
                 R7, #NR OF KSEGS
             !CONVERT SEG# TO KST INDEX!
03BE 000A
03C0 1906
            MULT RR6, #SIZEOF KST REC
             !OFFSET TO KST_REC!
03C2 0010
03C4 93F1
           PUSH
                 aR 15, R1
                         !SAVE SEGMENT#!
03C6 93F7
                 @R 15, R7
           PUSH
03C8 2101
           LD
                 R1, #KST_SEG_NO
03CA 0002
03CC 5F00
                 CALL
03CE 0000*
03D0 A10D
           LD
                 R13,R0
                         !¬KST!
                 R7, @R15
03D2 97F7
           POP
03D4 97F1
           POP
                 R1. OR15 !RETRIEVE SEGMENT#!
                 R13,R7 !ADD OFFSET TO KST
03D6 817D
           ADD
                                   BASE ADDR!
03D8 2106
           LD
                 R6, #NULL_SEG
03DA FFFF
03DC 4ADE
                 RL6, KST. M_SEG_NO(R13)
           CPB
03DE 000E
03E0 5E0E
                      THEN
           IF
                 ΕO
03E2 03EC*
03E4 2100
             LD
                   RO, #SEGMENT_NOT_KNOWN
03E6 001C
03E8 5E08
           ELSE
03EA 0426"
03EC 2106
                   R6, #FALSE
             LD
03EE 0000
03FO 4ADE
             CPB
                  RL6, KST.IN_CORE (R13)
03F2 0009
             IF EQ THEN
03F4 5E0E
03F6 0400°
               LD
                     RO, #SUCCEEDED
03F8 2100
```

```
03FA 0002
03FC 5E08
              ELSE
03FE 0426
0400 93FD
                       DR15,R13 !SAVE KST REC ADDR!
                 PUSH
0402 5F00
                 CALL GET_DBR_NUMBER !R1: (RET) DBR_NO!
0404 0000*
0406 97FD
                 POP
                       R13, @R15
0408 76D2
                       R2, KST. MM_HANDLE (R13)
                 LDA
040A 0000
040C 93FD
                 PUSH
                       DR15,R13 !SAVE SEG KST REC ADDR!
040E 5F00
                 CALL
                       MM_SWAP_OUT !R1:DBR_NO!
0410 0000*
                 !R2: ¬MM_HANDLE!
                 !RO: (RET) SUCCESS_CODE!
0412 5F00
                 CALL CONFINEMENT_CHECK
                 ! (RO:SUCCESS_CODE)!
0414 04281
0416 97FD
                 POP
                       R13, @R15
0418 0A08
                 CPB
                       RLO, #SUCCEEDED
041A 0202
041C 5E0E
                 IP
                      EQ
                             THEN
041E 0426
0420 4CD5
                   LDB
                         KST.IN_CORE (R13), #FALSE
0422 0009
0424 0000
                 FI
              ΡI
            FI
0426 9E08
            RET
0428
           END SM_SWAP_OUT
```

```
0428
          CONFINEMENT_CHECK
                                 PROCEDURE
         ! SERVICE ROUTINE TO VERIFY
         ! CONFINEMENT IS NOT VIOLATED
         ! WHEN MEM MGR SUCCESS CODE IS
         ! RETURNED TO SUPERVISOR.
         ! REGISTER USE:
                                          Ţ
           PARAMETERS
            RO:SUCCESS_CODE
         [*********************
          ENTRY
           IF RO
             CASE #LEAF_SEG_EXISTS THEN
0428 OB00
               CALL MONITOR
042A 000A
042C 5E0E
042E 0438
0430 5F00
0432 059A
0434 5E08
             CASE #NO LEAF_EXISTS THEN
               CALL MONITOR
0436 04B4 1
0438 OB00
043A 000B
043C 5E0E
043E 0448
0440 5F00
0442 059A
             CASE #ALIAS_DOES_NOT_EXIST THEN
0444 5E08
               CALL MONITOR
0446 04B41
0448 OB00
044A 0017
044C 5E0E
044E 0458*
0450 5F00
0452 059A
             CASE #NO_CHILD_TO_DELETE THEN
0454 5E08
               CALL MONITOR
0456 04B4 *
0458 OB00
045A 0014
045C 5E0E
045E 0468
0460 5F00
0462 059A
             CASE #G_AST_FULL THEN
0464 5E08
               CALL MONITOR
0466 04B41
0468 OB00
046A 000C
```

```
046C 5E0E
046E 0478*
0470 5F00
0472 059A
0474 5E08
             CASE #L_AST_FULL THEN
                CALL MONITOR
0476 04B4*
0478 OB00
047A 000D
047C 5E0E
047E 0488*
0480 5F00
0482 059A
0484 5E08
             CASE #LOCAL_MEMORY_FULL THEN
                CALL MONITOR
0486 04B4 ·
0488 OB00
048A 0010
048C 5E0E
048E 0498
0490 5F00
0492 059A
0494 5E08
              CASE #GLOBAL_MEMORY_FULL THEN
                CALL MONITOR
0496 04B4 •
0498 0B00
049A 0011
049C 5E0E
049E 04A8*
04A0 5F00
04A2 059A
04A4 5E08
             CASE #SEC_STOR_FULL THEN
                CALL MONITOR
04A6 04B4*
04A8 0B00
04AA 0015
04AC 5E0E
04AE 04B4 .
04B0 5F00
04B2 059A
            FI
04B4 9E08
           RET
04B6
          END CONFINEMENT_CHECK
```

END SEG_MGR

Appendix I

NON-DISCRETIONARY SECURITY LISTINGS

```
Z8000ASM 2.02
LOC OBJ CODE
                STMT SOURCE STATEMENT
       $LISTON $TTY
       NDS MODULE
       CONSTANT
         TRUE
                       :=1
         FALSE
                       :=0
       INTERNAL
         $SECTION ACC_CLASS_DCL !NOTE: IS AN OVERLAY,
                              IE NO ALLOCATION
                              OF MEMORY!
         $ABS 0
0000
         ACCESS_CLASS
                        RECORD [ LEVEL
                                       INTEGER
                                CAT
                                       INTEGER ]
       GLOBAL
       $SECTION NDS_PROC
0000
         CLASS_EQ
                             PROCEDURE
         ! PASSED PARAMETERS
           RR2 = CLASS1
           RR4 = CLASS2
         ! RETURNED
           R1 = CONDITION CODE
         ENTRY
0000 9042
          CPL
                 RR2,RR4
0002 5E0E
          IF
                 ΕQ
                        THEN
0004 000E
                  LD
                        R1. #TRUE
0006 2101
0008 0001
000A 5E08 ELSE
000C 0012'
                 LD
                        R1, #PALSE
000E 2101
0010 0000
          FI
0012 9E08
          RET
0014
         END CLASS_EQ
```

```
0014
          CLASS GE
                               PROCEDURE
          PASSED PARAMETERS
            RR2 = CLASS1
            RR4 = CLASS2
          ! RETURNED PARAMETER
            R1 = CONDITION CODE
          [ *******************
           ENTRY
0014 91F2
           PUSHL DR 15, RR2 ! PUSH CLASS1 ON STACK-
                                   -REFER BY ADDR!
0016 A1FD
                           ! CLASS1 ADDR !
                  R13,R15
           LD
            PUSHL @R15,RR4
0018 91F4
001A A1FE
                  R14.R15
                           ! CLASS2 ADDR !
            LD
001C 31E7
            LD
                  R7, R14 (#ACCESS_CLASS.CAT)
                            ! CAT2 IN R7 !
001E 0002
0020 45D7
            OR
                  R7, ACCESS_CLASS.CAT (R13)
                         !CAT1 OR CAT2, R7!
0022 0002
0024 4BD7
            CP
                  R7, ACCESS_CLASS.CAT (R13)
                        !CAT1 = (CAT1 OR CAT2) ?!
0026 0002
0028 5E0E
           IF EQ THEN
002A 0048*
002C 61D6
              LD
                  R6, ACCESS CLASS. LEVEL (R13)
                                      !LEVEL1!
002E 0000
                ! COMPARE LEVEL1 WITH LEVEL2 !
0030 4BE6
              CP
                  R6, ACCESS_CLASS. LEVEL (R14)
0032 0000
0034 5E01
                          THEN !LEVEL1 GE LEVEL2!
              IF
                 GΕ
0036 0040*
0038 2101
                   LD
                          R1. #TRUE
003A 0001
003C 5E08
              ELSE
003E 0044*
0040 2101
                   LD
                         R1, #FALSE
0042 0000
              FI
0044 5E08
            ELSE
0046 004C*
0048 2101
              LD
                 R1, #FALSE
004A 0000
            FI
004C 95F4
           POPL
                   RR4, @R15
004E 95F2
           POPL
                   RR2, DR15 ! RESTORE STACK!
0050 9808
           RET
0052
           END CLASS GE
        END NDS
```

Appendix J

MEMORY MANAGER LISTINGS

```
Z8000ASM 2.02
LOC
     OBJ CODE
                 STMT SOURCE STATEMENT
        $LISTON $TTY
       MM PROCESS
                      MODULE
       ! VERS. 1.9 !
       CONSTANT
          RETURN_TO_MONITOR := %A902 !HBUG REENTRY!
         COUNT := 10
         TIME := 500
         NR_OF_HOSTS
                      := 2
         G_AST_LIMIT := 10
         G_AST_FULL := 12
         FREE_ENTRY
                      := %EEEEEEEE
                       := %BBBB
         TRUE
                       := %CCCC
         FALSE
         SPACE
                       := %20
                       := %2D
         DASH
          IO_MGR
                       := %60
                      := %40
          FILE_MGR
          MEM_MGR
                       := %00
          FM_ENTRY
                       := %4A00
         IO_ENTRY
                       := %4 E0 0
         CREATE_ENTRY_CODE := 50
         INVALID_MMGR_CODE := 60
          DELETE_ENTRY_CODE
                           := 51
         ACTIVATE_SEG_CODE := 52
         DEACTIVATE_SEG_CODE:= 53
         SWAP_IN_SEG_CODE := 54
          SWAP_OUT_SEG_CODE
                           := 55
                            := 2
          SUCCEEDED
                            := 1
          STK SIZE
          TOP SECRET
                            := 4
```

:= 3

:= 2

:= 1

:= 0

:= 1

SECRET

EMPTY

CRYPTO

UNCLASS

CONFIDENTIAL

```
NATO
                    := 2
  NUCLEAR
                     := 4
TYPE
  ADDRESS
                WORD
  H_ARRAY
                ARRAY[3 WORD]
  G_AST_REC
                RECORD
   [UNIQUE_ID
                  LONG
    GLOBAL_ADDR
                  ADDRESS
    P_L_ASTE_NO
                  WORD
    FLAG_BITS
                   WORD
    G_ASTE_PAR
                   WORD
    NO ACT IN MEM WORD
    NO_ACT_DEP
                   BYTE
    SIZE1
                   BYTE
    PG_TBL_LOC
                  ADDRESS
    ALIAS_TBL_LOC ADDRESS
    SEQUENCER
                  LONG
    EVENT1
                  LONG
    EVENT 2
                  LONG
   1
EXTERNAL
  SIGNAL
                  PROCEDURE
  TIAW
                  PROCEDURE
  TC_INIT
                  PROCEDURE
  GET_CPU_NO
                  PROCEDURE
  CREATE_PROCESS PROCEDURE
  SNDCHR
                  PROCEDURE
  SNDMSG
                  PROCEDURE
  SNDCRLF
                  PROCEDURE
  G_AST_LOCK
                  WORD
  G_AST ARRAY[G_AST_LIMIT G_AST_REC]
GLOBAL
$SECTION MM_DATA
   MM_ENTRY LABEL
```

INTERNAL

```
! * * * * MESSAGES * * * * * !
0000 08
         28
              IO ARRAY [* BYTE] := "%08(FOR IO)"
0002 46
         4 F
0004 52
        20
0006 49
         4F
0008 29
0009 08
              PM ARRAY [* BYTE] := "%08(FOR FM)"
        28
000B 46
         4P
000D 52
         20
000F 46
         4 D
0011 29
0012 12
        4 B
               MM_MSG_1
                 ARRAY [* BYTE] := "%12KERNEL = SIGNALLER"
         52
0014 45
0016 4E
         45
0018 4C
         20
001A 3D
         20
001C 53
         49
001E 47
         4E
0020 41
         4C
        45
0022 4C
0024 52
0025 10 4D
               CREATE_MSG
                 ARRAY [ * BYTE ]: = "%10MM: CREATE_ENTRY"
0027 4D
        3 A
0029 20 43
002B 52
         45
002D 41
         54
         5 F
002F 45
0031 45
         4 E
0033 54
         52
0035 59
               DELETE_MSG
0036 10
        4 D
                 ARRAY [* BYTE] := "%10MM: DELETE_ENTRY"
0038 4D
         3 A
003A 20
         44
         4C
003C 45
003E 45
         54
0040 45
         5F
0042 45
         4 E
0044 54
         52
0046 59
```

```
0047 OC
        4 D
               ACTIVATE_MSG
                 ARRAY [* BYTE] := "%OCMM: ACTIVATE"
0049 4D
          3 A
004B 20
          41
004D 43
          54
004F 49
          56
0051 41
          54
0053 45
0054 OE
               DEACTIVATE_MSG
          4 D
                 ARRAY [ * BYTE ] := "%OEMM: DEACTIVATE"
0056 4D
          3 A
0058 20
          44
005A 45
          41
005C 43
          54
005E 49
          56
0060 41
          54
0062 45
0063 OB
         4 D
               SWAP_IN_MSG
                 ARRAY [* BYTE] := "%OBMM: SWAP_IN"
0065 4D
          3A
0067 20
          53
0069 57
          41
006B 50
         5 F
006D 49
         4 E
006F 0C
         4 D
               SWAP_OUT_MSG
                 ARRAY [* BYTE] := "%OCMM: SWAP_OUT"
0071 4D
          3 A
0073 20
          53
0075 57
         41
0077 50
         5 F
0079 4F
         55
007B 54
007C 0C
         49
               ERROR MSG
                 ARRAY [ * BYTE] := "%OCINVALID CODE"
007E 4E
         56
0080 41
         4C
0082 49
         44
0084 20
          43
0086 4F
         44
0088 45
0089 02
          00
               RET VALUES
                 ARRAY [* BYTE] := [2,0,0,0,0,16,0,17,0,3,0,
008B 00
         00
008D 00
         10
008F 00
         11
0091 00
         03
0093 00
         01
0095 00
         30
0097 00
         00
                                                     1,0,48,0,0]
099A
               MM_MSG_ARRAY
                                ARRAY
                                         [ 8 WORD ]
OOAA
               SENDER
                                 WORD
```

```
$ABS 0
             INO MEMORY ALLOCATED; USED
              FOR PARAMETER TEMPLATE ONLY!
0000
             ACTIVATE_ARG RECORD
          CODE
                         WORD
             DBR
                         WORD
             HANDLE
                        H ARRAY
                        BYTE
             ENTRY_NO
             SEG_NO
                        BYTE
           1
            $ABS 0
             ! NO MEMORY ALLOCATED: USED
              FOR PARAMETER TEMPLATE ONLY!
0000
             RET_VAL
                            RECORD
             CODE1
                        BYTE
          ſ
             FILLER
                        BYTE
             MM_HANDLE H_ARRAY
             CLASS
                        LONG
             SIZE
                         WORD
             FILLER 1
                         WORD
           1
             $ABS 0
             ARG_LIST RECORD
0000
             REG
          ſ
                      ARRAY[13 WORD]
             IC
                       WORD
             CPU_ID
                       WORD
             SAC
                       LONG
             PRI
                       WORD
             USR_STK
                      WORD
             KER STK
                      WORD
           1
```

\$SECTION MM_PROC

```
0000
                               PROCEDURE
            MM MAIN
           ENTRY
           MM_ENTRY:
            ! INITIALIZE G AST !
                 G_AST_LOCK
0000 4D08
            CLR
0002 0000*
                 R2, #1
0004 2102
            LD
0006 0001
0008 2101
           LD R1, #0
000A 0000
000C 1404
           LDL
                 RR4, #FREE_ENTRY
OOOE EEEE
0010 EEEE
            DO
0012 5D14
              LDL G_AST.UNIQUE_ID(R1), RR4
0014 0000*
0016 A920
             INC R2. #1
0018 0B02
                  R2, #G_AST_LIMIT
             CP
001A 000A
001C 5E02
              IF GT !END OF G_AST! THEN
001E 00241
0020 5E08
             EXIT FI
0022 002A'
             ADD R1, #SIZEOF G AST REC
0024 0101
0026 0020
0028 E8F4
          OD
                  ! RESERVE FIRST ENTRY IN
                           G AST FOR ROOT !
002A 2101
           LD
                 R1, #0
002C 0000
002E 1404
            LDL
                 RR4, #-1
0030 FFFF
0032 FFFF
0034 5D14
            LDL
                G_AST.UNIQUE_ID(R1), RR4
0036 0000*
0038 5F00
            CALL GET_CPU_NO !RETURNS:
003A 0000*
                            R1: CPU #
                            R2: # VP'S!
003C 93F1
                  DR15, R1 !SAVE CPU #!
            PUSH
003E 5F00
            CALL TC_INIT
0040 0000*
             ! USER/HOST # !
0042 210D
            LD R13, #0
0044 0000
             ! INITIALIZE USERS !
            DO
0046 A9D0
             INC R13, #1
```

```
0048 OBOD
              CP R13, #NR_OF_HOSTS
004A 0002
              IF GT !ALL HOSTS INITIALIZED!
004C 5E02
              THEN EXIT
004E 0054*
0050 5E08
0052 00B8*
              FI
               ! CREATE FM PROCESS !
                   RO, @R15 ! RESTORE CPU #!
0054 21F0
              LD
0056 030F
              SUB
                   R15, #SIZEOF ARG_LIST
0058 0028
              !SETS ARGUMENT LIST IN STACK!
005A A1F1
              LD R1, R15
005C 6F10
                   ARG_LIST.CPU_ID(R1), R0
              LD
005E 001C
              !LOAD INITIAL REGISTER PARAMETERS
               FOR FM PROCESS (SIMULATED)
               R13 DENOTES USER # !
0060 5C19
              LDM ARG LIST.REG(R1), R2, #13
0062 020C
0064 0000
              LD R2. #FM ENTRY
0066 2102
0068 4A00
006A 6F12
                   ARG LIST.IC(R1), R2
              LD
006C 001A
006E 2102
              LD R2. #SECRET
0070 0003
0072 8D38
              CLR
                   R 3
0074 0503
              OR
                   R3, #CRYPTO
0076 0001
0078 5D12
              LDL ARG LIST.SAC (R1), RR2
007A 001E
007C 4D15
              LD ARG LIST.PRI (R1), #2
007E 0022
0080 0002
                   ARG_LIST.USR_STK (R1), #STK_SIZE
0082 4D15
              LD
0084 0024
0086 0001
                   ARG LIST.KER STK (R1) , #STK_SIZE
0088 4D15
              LD
008A 0026
008C 0001
008E A11E
              LD R14, R1
0090 93FD
              PUSH DR15, R13
              CALL CREATE_PROCESS ! R14: ARG PTR!
0092 5F00
0094 0000*
0096 97FD
              POP R13, @R15
               ! CREATE IO PROCESS !
```

```
0098 A1F1 LD R1, R15 !RESTORE ARGUMENT PTR!
               !LOAD INITIAL REGISTER PARAMETERS
               FOR IO PROCESS (SIMULATED)
               R13 DENOTES USER # !
009A 5C19
             LDM ARG_LIST.REG(R1), R2, #13
009C 020C
009E 0000
00A0 2102
             LD R2, #IO ENTRY
00A2 4E00
00A4 6F12
             LD ARG LIST.IC(R1), R2
00A6 001A
00A8 A11E
             LD R14, R1
00AA 93FD
             PUSH DR15, R13
00AC 5F00
             CALL CREATE PROCESS ! R14: ARG PTR!
00AE 0000*
00B0 97FD
             POP R13, 0R15
00B2 010F
             ADD R15, #SIZEOF ARG_LIST
00B4 0028
00B6 E8C7
          OD
            ! REMOVE CPU # FROM STACK !
00B8 97F0
           POP RO, DR15
           DO !** DO FOREVER **!
00BA 7608
             LDA
                    R8,MM MSG ARRAY O
00BC 009A1
00BE 5F00
            CALL WAIT
00C0 0000*
            LD SENDER, R1 !SAVE SIGNALING PROC #!
00C2 6F01
00C4 00AA*
00C6 2103
             LD
                    R3,#50
00C8 0032
00CA 5F00
             CALL MM_PRINT_BLANKS
00CC 030C*
00CE 2102
                    R2, #MM_MSG_1
             LD
00D0 00121
00D2 5F00
             CALL
                     SNDMSG
00D4 0000*
00D6 6101
             LD R1, SENDER
00D8 00AA*
                    R 1
             ΙF
00DA 0B01
               CASE #IO MGR THEN LD R2,#IO
00DC 0060
00DE 5E0E
00E0 00EE'
00E2 2102
00E4 0000°
                 CALL SNDMSG
00E6 5F00
00E8 0000*
00EA 5E08
               CASE #FILE_MGR THEN LD R2, #FM
OOEC OOPE'
00EE 0B01
00F0 0040
00F2 5E0E
```

00F4 00F6	00FE' 2102		
	00091		
OOFA	5F00	CALI	SNDMSG
OOFC	0000*		
		FI	
OOFE	5F00	CALL	MM_DELAY
0100	02D8 •		
0102	5F00	CALL	SNDCRLF
0104	0000*		
0 106	2103	LD	R3,#50
0108	0032		
010A	5F00	CALL	MM_PRINT_BLANKS
010C	030C1		
010E		LD	R1, MM_MSG_ARRAY O
0110	009A		

```
IF R1
0112 0B01
               CASE #CREATE_ENTRY_CODE THEN
0114 0032
0116 5E0E
0118 0122
011A 5F00
                CALL CREATE_ENTRY
011C 019E'
011E 5E08
             CASE #DELETE_ENTRY_CODE THEN
0120 0176
0122 0B01
0124 0033
0126 5E0E
0128 0132
012A 5F00
                CALL DELETE_ENTRY
012C 01AC
012E 5E08
              CASE #ACTIVATE SEG CODE THEN
0130 0176
0132 0B01
0134 0034
0136 5E0E
0138 0142
013A 5F00
                CALL ACTIVATE
013C 01BA
013E 5E08
            CASE #DEACTIVATE_SEG_CODE THEN
0140 0176
0142 0B01
0144 0035
0146 5E0E
0148 0152
014A 5F00
                CALL DEACTIVATE
014C 029E
014E 5E08
              CASE #SWAP_IN_SEG_CODE THEN
0150 0176
0152 0B01
0154 0036
0156 5E0E
0158 0162
015A 5F00
               CALL SWAP_IN
015C 02AC
015E 5E08
              CASE #SWAP_OUT_SEG_CODE THEN
0160 0176
0162 0B01
0164 0037
0166 5E0E
0168 0172
016A 5F00
                CALL SWAP_OUT
016C 02CA*
016E 5E08
            ELSE
0170 0176
0172 2102
             LD R2, #ERROR_MSG
0174 007C*
             FI
```

```
0176 5F00
           CALL SNDMSG
0178 0000*
017A 5F00
            CALL MM_DELAY
017C 02D8
017E 5F00
           CALL SNDCRLF
0180 0000*
0182 2103
            LD
                 R3.#75
0184 004B
0186 5F00
            CALL MM_PRINT_LINE
0188 02F4
018A 5F00
            CALL
                   SNDCRLF
018C 0000*
            ! ** SIGNAL (SENDER, DONE) **!
018E 6101
                R1, SENDER
            LD
0190 00AA
0192 7608
           LDA
                   R8,MM_MSG ARRAY O
0194 009A
0196 5F00
           CALL SIGNAL
0198 0000*
019A E88F OD ! ** REPEAT FOREVER **!
019C 9E08
          RET
019E
         END MM MAIN
019E
         CREATE_ENTRY PROCEDURE
         ENTRY
019E 7608
          LDA R8, MM_MSG_ARRAY O
01A0 009A .
01A2 0C85 LDB @R8,#SUCCEEDED
01A4 0202
01A6 2102 LD R2, #CREATE_MSG
01A8 0025
01AA 9E08
          RET
01AC
         END CREATE_ENTRY
O 1 AC DELETE_ENTRY PROCEDURE
         ENTRY
               R8,MM_MSG_ARRAY 0
01AC 7608
          LDA
01AE 009A
01B0 0C85 LDB @R8, #SUCCEEDED
01B2 0202
         LD R2, #DELETE MSG
01B4 2102
01B6 0036
01B8 9E08
         RET
01BA
      END DELETE_ENTRY
```

```
01BA ACTIVATE
                                   PROCEDURE
                   ! R8: ARGUMENT PTR !
          ENTRY
01BA 7608
           LDA R8, MM_MSG_ARRAY O
01BC 009A'
O1BE 6182 LD R2, ACTIVATE_ARG. HANDLE 2 (R8)
                                    !UNIQUE ID!
01C0 0008
01C2 8D38
          CLR
                 R3
01C4 608B
                RL3, ACTIVATE_ARG.ENTRY_NO(R8)
          LDB
01C6 000A
01C8 030F
         SUB R15, #SIZEOF RET_VAL
01CA 0010
01CC A1F8
          LD
                 R8, R15
01CE 2100
                 RO, #FALSE
           LD
01D0 CCCC
01D2 2101
          LD R1, #0 !G_AST INDEX!
01D4 0000
01D6 2104 LD R4, #1 !NR OF ENTRIES SEARCHED!
01D8 0001
           SEARCH_G_AST:
           DO
             CPL RR2, G_AST.UNIQUE_ID(R1)
01DA 5012
01DC 0000*
01DE 5E0E
            IF EO !SEGMENT IS ACTIVE! THEN
01E0 01EA
01E2 2100
             LD RO, #TRUE
01E4 BBBB
01E6 5E08
             EXIT FROM SEARCH_G_AST
01E8 01FE*
             PΙ
01EA A940
             INC 84, #1
                  R4, #G_AST_LIMIT
01EC 0B04
             CP
01EE 000A
01F0 5E02
            IF GT !END OF G_AST! THEN
01F2 01F8
01F4 5E08
             EXIT FROM SEARCH_G_AST
01F6 01FE
             FI
01F8 0101
             ADD R1, #SIZEOF G_AST_REC
01FA 0020
01FC E8EE
         OD
          CP RO, #FALSE
01FE 0B00
0200 CCCC
           IF EQ !SEGMENT NOT ACTIVE!
0202 5E0E
             THEN
0204 0266
0206 2100
            LD RO, #1
0208 0001
020A 2101
            LD
                  R1. #0
```

```
020C 0000
             FIND_FREE_ENTRY:
               DO
020E 1404
                 LDL RR4, #FREE ENTRY
0210 EEEE
0212 EEEE
0214 5014
                 CPL
                      RR4, G_AST.UNIQUE ID (R1)
0216 0000*
0218 5E0E
                IF EQ !ENTRY IS AVAILABLE! THEN
021A 0220'
021C 5E08
                   EXIT FROM FIND PREE ENTRY
021E 0234'
                 FI
                 INC
0220 A900
                      RO, #1
0222 OB00
                 CP
                      RO, #G_AST_LIMIT
0224 000A
0226 5E02
                 IF GT ! END OF G_AST! THEN
0228 022E
022A 5E08
                  EXIT FROM FIND_FREE_ENTRY
022C 02341
                 PΙ
022E 0101
                 ADD
                      R1, #SIZEOF G_AST_REC
0230 0020
0232 E8ED
               OD
                      RO, #G_AST_LIMIT
0234 OB00
               CP
0236 000A
                  LE !FOUND FREE ENTRY!
               IF
0238 5E0A
                   THEN
023A 025C
023C 5D12
                    LDL G AST. UNIQUE_ID(R1), RR2
023E 0000*
                ! ZERO ALL EVENT DATA ENTRIES !
0240 1404
                LDL RR4, #0
0242 0000
0244 0000
                LDL G AST. SEQUENCER (R1), RR4
0246 5D14
0248 0014*
024A 5D14
                 LDL G_AST.EVENT1(R1), RR4
024C 0018*
024E 5D14
                 LDL G AST. EVENT2 (R1), RR4
0250 001C*
                 LDB RET_VAL.CODE1 (R8) . #SUCCEEDED
0252 4C85
0254 0000
0256 0202
0258 5E08
              ELSE
025A 0262
                LDB RET VAL.CODE1 (R8) , #G_AST_FULL
025C 4C85
025E 0000
0260 OCOC
               FI
0262 5E08
             ELSE !SEGMENT ACTIVE!
```

```
0264 026C1
0266 4C85
            LDB RET_VAL.CODE1 (R8) . #SUCCEEDED
0268 0000
026A 0202
           FΙ
026C 5D82
           LDL
                 RET_VAL.MM_HANDLE O (R8), RR2
026E 0002
0270 6F81
           LD
                 RET VAL.MM HANDLE 2 (R8), R1
0272 0006
0274 1404
           LDL
                 RR4, #%30001
0276 0003
0278 0001
027A 5D84
           LDL RET_VAL. CLASS (R8) , RR4
027C 0008
027E 4D85
           LD
                 RET_VAL.SIZE(R8), #1
0280 000C
0282 0001
0284 7689
           LDA R9, RET_VAL (R8)
0286 0000
0288 7608
           LDA R8, MM MSG ARRAY O
028A 009A*
028C 2102
                 R2, #16
           LD
028E 0010
0290 BA91 LDIRB @R8, @R9, R2
0292 0280
0294 2102
                 R2, #ACTIVATE_MSG
           LD
0296 0047
0298 010F
           ADD R15, #SIZEOF RET_VAL
029A 0010
029C 9E08
          RET
         END ACTIVATE
029E
```

```
029E DEACTIVATE PROCEDURE
         ENTRY
029E 7608
         LDA R8, MM_MSG_ARRAY O
02A0 009A
02A2 0C85 LDB @R8,#SUCCEEDED
02A4 0202
02A6 2102 LD R2, #DEACTIVATE_MSG
02A8 0054 ·
02AA 9E08 RET
02AC
      END DEACTIVATE
O2AC SWAP_IN
                              PROCEDURE
         ENTRY
02AC 2102 LD R2, #%FF30
02AE FF30
02B0 3B26 OUT %FFD2, R2
02B2 FFD2
02B4 7608
         LDA R8, MM_MSG_ARRAY
02B6 009A ·
02B8 5F00 CALL WAIT !R8:MSG ARRAY!
02BA 0000*
02BC 7608 LDA R8, MM_MSG_ARRAY 0
02BE 009A'
02CO 0C85 LDB @R8,#SUCCEEDED
02C2 0202
02C4 2102 LD R2, #SWAP_IN_MSG
02C6 0063
02C8 9E08 RET
02CA
        END SWAP_IN
```

```
02CA SWAP_OUT
                         PROCEDURE
         ENTRY
02CA 7608
               R8,MM_MSG_ARRAY 0
          LDA
02CC 009A*
02CE 0C85 LDB @R8,#SUCCEEDED
02D0 0202
02D2 2102 LD R2, #SWAP_OUT_MSG
02D4 006F*
02D6 9E08
         RET
02D8
         END SWAP OUT
02D8
        MM DELAY
                              PROCEDURE
        ! PRODUCES 2 SEC DELAY
        ! ***************************
         ENTRY
02D8 2102
                R2, #COUNT
         LD
02DA 000A
02DC 2101
         LD
                R1, #TIME
02DE 01F4
          DO
02E0 0B02
          CP R2, #0
02E2 0000
         IF EQ THEN EXIT FI
02E4 5E0E
02E6 02EC
02E8 5E08
02EA 02F2*
02EC AB20
           DEC R2
02EE 7B1D
           MREQ
                 R1
02F0 E8F7
         OD
02F2 9E08
         RET
02F4
        END MM_DELAY
```

```
02F4
         MM_PRINT LINE
                             PROCEDURE
        ! PRINTS LINE LENGTH
        ! SPEC IN R3.
        ENTRY
02F4 C82D
          LDB
                RLO, #DASH
          DO
02F6 0B03
           CP
                 R3, #0
02F8 0000
02FA 5E0E
           IF EQ THEN EXIT FI
02FC 03021
02FE 5E08
0300 030A4
0302 5F00
           CALL SNDCHR
0304 0000*
0306 AB30
            DEC
                  R3
0308 E8F6
          OD
030A 9E08
         RET
030C
         END MM_PRINT_LINE
030C
         MM_PRINT_BLANKS
                             PROCEDURE
        ! PRINTS NUMBER OF
                                     •
        ! BLANKS SPEC IN R3.
                                     Į
        ENTRY
030C C820
                RLO, #SPACE
          LDB
          DO
030E 0B03
                 R3, #0
           CP
0310 0000
0312 SEOE
           IF EQ THEN EXIT FI
0314 031A'
0316 5E08
0318 03221
031A 5F00
           CALL SNDCHR
031C 0000*
031E AB30
           DEC
                 R3
0320 E8F6
          OD
0322 9E08
          RET
         END MM_PRINT_BLANKS
0324
      END MM_PROCESS
```

LIST OF REFERENCES

- 1. Advanced Micro Computers, AM96/4116 AMZ8000 16-bit Monoboard Computer, Users's Manual, 1980.
- Coleman, A. R., <u>Security Kernel Design for a Microprocessor-Based</u>, <u>Multilevel</u>, <u>Archival Storage System</u>, MS Thesis, Naval Postgraduate School, December 1979.
- 3. Denning, D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, V. 19, p 236-242, May 1976.
- 4. Dijkstra, E. W., "The Humble Programmer,"

 <u>Communications of the ACM</u>, V. 15, No. 10, p. 859-865,
 October 1972.
- 5. Gary, A. V. and Moore, E. E., <u>The Design and</u>
 <u>Implementation of the Memory Manager for a Secure</u>
 <u>Archival Storage System</u>, MS Thesis, Naval Postgraduate School, June, 1980.
- 6. Madnick, S. E. and Donovan, J. J., Operating Systems, McGraw Hill, 1974.
- 7. O'Connell, J. S. and Richardson, L. D., <u>Distributed</u>
 <u>Secure Desqn for a Multi-microprocessor Operating</u>
 <u>System</u>, MS Thesis, Naval Postgraduate School, June
 1980.
- 8. Organick, E. J., The <u>Multics System</u>: An <u>Examination of</u>
 <u>Its Structure</u>, MIT Press, 1972.
- 9. Parks, E. J., <u>The Design of a Secure File Storage</u>

 <u>System</u>, MS Thesis, Naval Postgraduate School, December 1979.
- 10. Reed, P. D., <u>Processor Multiplexing in a Layered</u>

 <u>Operating System</u>, MS Thesis, Nassachusetts Institute of Technology, MIT LCS/TR-167, 1979.
- 11. Reed, P. D. and Kanodia, R. K., "Sycnhronization With Eventcounts and Sequencers," <u>Communications of the ACM</u>, V. 22, No. 2, pp. 115-124, February 1979.

- 12. Reitz, S. L.., An Implementation of Multiprogramming and Process Management for a Security Kernel Operating System, MS Thesis Naval Postgraduate School, June 1980.
- 13. Riggins, C., "When No Single Language Can Do the Job, Make it a Language-Family Matter," <u>Electronics Design</u>, February 15, 1979.
- 14. Saltzer, J. H., <u>Traffic Control in a Multiplexed</u>

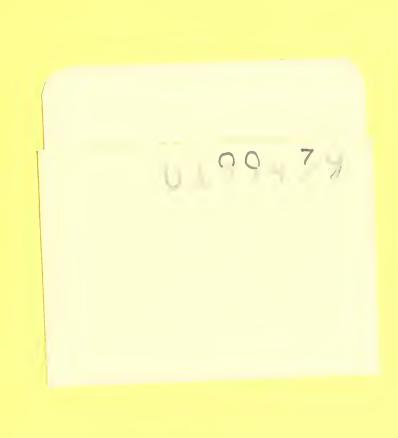
 <u>Computer System</u>, Ph.D. Thesis, Massachusetts Institute of Technology, 1966.
- 15. Schell, Lt. Col. R. R., "Computer Security: the Achilles Heel of the Electronic Air Forece?," Air University Review, V. 30, No. 2, pp. 16-33, January 1979.
- 16. Schell, Lt. Col. R. R., "Security Kernels: A Methodical Design of System Security," <u>USE Technical Papers</u> (Spring Conference, 1979) pp. 245-250, March 1979.
- 17. Schell, R. R. and Cox, L. A., <u>Secure Archival Storage</u>
 System, <u>Part I -- Design</u>, Naval Postgraduate School,
 NPS52-80-002, March 1980.
- 18. Schroeder, M.D., "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, V. 15, No. 3, pp. 157-170, March 1972.
- 19. Strickler, A. R., <u>Implementation of Process Management</u>
 <u>for a Secure Archival Storage System</u>, MS Thesis, Naval
 Postgraduate School, March 1981.
- 20. Wells, J. T., <u>Implementation of Sequent Management for a Secure Archival Storage Sytem</u>, MS Thesis, Naval Postgraduate School, Septemeber 1980.
- 21. Zilog, Inc., Z8000 PLZ/ASM Assembly Language Programming Manual, 03-3055-01, Revision A, April 1979.
- 22. Zilog, Inc., Z8001 CPU Z8002 CPU, Preliminary Product Specification, March 1979.
- 23. Zilog, Inc., Z8010 MMU Memory Management Unit, Preliminary Product Specification, October 1979.

INITIAL DISTRIBUTION LIST

		No.	Copies
1.	Defense Documentation Center ATTN:DDC-TC Cameron Station Alexandria, Virginia 22314		2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940		2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940		2
4.	National Security Agency Attn.: Col. Roger R. Schell C1 Fort George Meade, Maryland 20755		5
5.	Lyle A. Cox, Jr., Code 52C1 Department of Computer Science Naval Postgraduate School Monterey, California 93940		4
6.	Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217		1
7.	John P.L. Woodward The MITRE Corporation P.O. Box 208 Bedford, Massachusetts 01730		1
8.	Digital Equipment Corporation Attn: Mr. Donald Gaubatz 146 Main Street ML 3-2/E41 Maynard, Massachusetts 01754		1

9.	University of Southwestern Louisiana	7
	P.O. Box 44330	
	Lafayette, Louisiana 70504	
1.0	ICDD Comp Dokon Codo 27	
10.	LCDR Gary Baker, Code 37	1
	COMRESPATWINGPAC Code 32	
	NAS Moffett Field, California 94035	
	and noticed recta, earliefuld 14033	
11.	LCDR John T. Wells	1
	P.O. Box 366	
	Waynesboro, Mississippi 39367	
12	James P. Anderson Co.	1
120	Box 42	
	Fort Washington, Pennsylvania 19034	
13.	I. Larry Avrunin, Code 18	1
	DTNSRDC	
	Bethesda, Maryland 20084	
14.	Gerald B. Blanton	1
	242 San Carlos Way	
	Novato, Calif. 94947	
15.	Intel Corporation	1
	Attn: Mr. Robert Childs	
	Mail Code: SC4-490 3065 Bowers Avenue	
	Santa Clara, California 95051	
	Junta Cidia, California 7505.	
16.	Dr. J. McGraw	1
	U.C L.L.L. (1-794)	
	P.O. Box 808	
	Livermore, California 94550	
17.	M. George Michael	1
	U.C L.L.L. (L-76)	
	P.O. Box 808	
	Livermore, California 94550	
18.	Robert Montee	1
10.	Director, Long Range Systems Planning	
	Honeywell, MN 12-2276	
	Honeywell Plaza	
	Minneapolis, Minn. 55408	

19.	Research and Developement Digital Equipment Corp. 146 Main Street Maynard, Mass. 01754	1
20.	David P. Reed MIT Lab for Computer Science 545 Tech Sq Cambridge, Mass. 02139	1
21.	Capt. Anthony R. Strickler HQ Command, US Army Signal Center & Ft. Gordon Ft. Gordon, Georgia 30905	1 (WOU5AA)
22.	Lt. W. J. Wasson Naval Electronics Systems Command Headquarters, PME 124 Washington D. C. 20360	1
23.	S. H. Wilson Code 7590 Naval Research Lab Washington D. C. 20375	1
24.	Maj. Robert Yingling Box 6227 APO New York, New York 09012	1
25.	LCDR William R. Shockley Code 52Sp Department of Computer Science Naval Postgraduate School Monterey, California 93940	40
26.	LCDR Ronald W. Modes Code 52Mf Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
27.	S. L. Perdue Code 52 Department of Computer Science Naval Postgraduate School	5





U197439

